21/22

NOTE DI SOFTWARE

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FPDM-119
RNSW-115

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA

Honeywell
Honeywell Information Systems Italia

Note di software.
è un bollettino trimestrale, edito a cura del Centro di Ricerca e Progettazione della Honeywell Information Systems Italia e dell'Istituto di Cibernetica dell'Università di Milano.

Note di software.
intende costituire uno strumento per la rapida e informale circolazione di informazioni, idee ed esperienze nell'area della tecnologia del software. In particolare, intende stimolare il dibattito sulle metodologie orientate alla diminuzione del rapporto costo/qualità dei programmi, e sugli strumenti atti ad automatizzare l'attività di produzione del software in ogni suo aspetto.

La collaborazione a **Note di software** è libera.
In particolare, si sollecitano contributi nella forma di brevi monografie e bibliografie essenziali commentate sui seguenti temi: tecniche e strumenti per la programmazione strutturata e modulare, metodologie e strumenti per il testing e il debugging dei programmi, documentazione del software, aspetti organizzativi e gestionali nella costruzione di sistemi software di grandi dimensioni.

Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Comitato di Redazione (HISI - Via ai Laboratori Olivetti - Pregnana Milanese), o di inviare direttamente il dattiloscritto in triplice copia.
I contributi non dovrebbero superare le quattromila parole.

I lavori accettati per la pubblicazione non vengono revisionati da un comitato tecnico. Pertanto, ogni contributo pubblicato riflette le opinioni dei suoi autori, e non necessariamente quelle del Comitato di Redazione.
Eventuali revisioni saranno effettuate di comune accordo fra il Comitato di Redazione e i proponenti.

Note di software.
Viene distribuito a chi ne faccia richiesta alla redazione.

Direttore responsabile: Paolo Ghelfi
Comitato di Redazione: Roberto Polillo - Wanda Anselmetti - Stefania Bandini
Redazione: Franco Soresini

NOTE DI SOFTWARE

21/22

Dicembre 1983

FPDM-119

Indice:

QUESTO NUMERO... pag. 1

DISCUSSIONI:

– PROTOTIPIZZAZIONE: SPUNTI ED ESPERIENZE DA MILANO, raccolte da G. Degli Antoni » 3

CONTRIBUTI TECNICI:

– GLI STANDARDS DI DOCUMENTAZIONE TECNICA NELLA PRODUZIONE INDUSTRIALE DI SOFTWARE, di A. Cicu, C. Cucciati, M. Maiocchi » 6

– STRUMENTI GRAFICI DI SOFTWARE DESIGN, di F. Capozza, F. Esposito, C. Gallo » 53

– L'APPROCCIO DI JACKSON AL SYSTEM DEVELOPMENT (JSD): ALCUNE NOTE, di C. Simone » 79

– OBJ, UN LINGUAGGIO DI PROGRAMMAZIONE BASATO SULLA LOGICA EQUAZIONALE, di G. Mauri » 84

– ESPERIENZE DI SVILUPPO DI UN SISTEMA INTERATTIVO PER IL PROGETTO CONCETTUALE DI ASPETTI DINAMICI DI BASI DI DATI, di R. Bertocchi, V. De Antonellis, R. Giovacchini » 92

BIBLIOGRAFIE:

– GLI EXPERT SYSTEMS, di S. Bandini, M. Gallanti » 101

RECENSIONI:

– T.G. Lewis, "Software Engineering – Analysis and Verification", curata da M. Torelli » 105

QUESTO NUMERO...

Le metodologie di progettazione, gli strumenti linguistici per la specifica del disegno del Software, e gli standards di documentazione del Software sono gli argomenti principali di questo numero doppio di NOTE DI SOFTWARE.

Talvolta il software di qualità è spesso software di autore, ovvero è realizzato da poche persone di cui si conosce il nome e di cui è nota la competenza. Ciò suggerisce di considerare, come possibile metodologia di programmazione, il modo impiegato in ambiti universitari, dove essenzialmente poche persone con una limitata divisione del lavoro cooperano alla realizzazione di prodotti spesso rifacendo completamente parti piuttosto che tentare di migliorarle allorchè vengono valutate come inadeguate.
Nel primo lavoro di questo numero, G. Degli Antoni propone una discussione su tali metodologie, cui spesso viene dato il nome di metodologie di prototipizzazione, inquadrando, al contempo, numerose esperienze effettuate, a questo proposito, da numerose persone nell'ambito dell'Università di Milano.

Nel secondo lavoro, A. Cicu, C. Cucciati e M. Maiocchi presentano un ampio inquadramento delle problematiche relative alla documentazione tecnica nella produzione industriale di software, e un insieme dettagliato di standards di documentazione, collegato a tutte le fasi del ciclo di vita del software.

Il terzo lavoro, di F. Capozza, C. Gallo e F. Esposito, d'altro canto, presenta un'ampia rassegna sugli strumenti grafici più diffusi per la rappresentazione del "software design". Vengono presentati, con esempi, una quindicina di strumenti diversi, dai diagrammi di Jackson, ai diagrammi SADT, alle reti di Petri, alle tavole decisionali, e così via, di ciascuno dei quali gli autori discutono le finalità specifiche.

Nel quarto lavoro, C. Simone presenta il metodo proposto da M. Jackson per il System Development, basato sulla costruzione di modelli del sistema da realizzare e dell'ambito in cui dovrà operare. Dopo un suo inquadramento all'interno delle problematiche dell'ingegneria del software vengono delineate le fasi principali che lo costituiscono utilizzando un semplice esempio.
Alcune note critiche mettono in evidenza da un lato gli aspetti innovativi e dall'altro alcune carenze nella modellizzazione della concorrenza.

OBJ, un linguaggio di programmazione ad altissimo livello di tipo funzionale, proposto da J. A. Goguen, ha caratteristiche di estremo interesse per l'utilizzo nell'ambito di una metodologia di programmazione avanzata, tra cui meccanismi di definizione di tipi di dati anche parametrici da parte dell'utente, facilitazioni per la programmazione gerarchica e modulare, strumenti di debugging interattivi molto potenti. Nel quinto lavoro, G. Mauri presenta le caratteristiche concettuali fondamentali di OBJ, senza entrare nei dettagli delle teorie e matematiche sulla base delle quali tale linguaggio è stato sviluppato.

Infine, R. Bertocchi, V. De Antonellis e R. Giovacchini descrivono alcune esperienze effettuate nello sviluppo di un sistema interattivo per il progetto concettuale di aspetti dinamici di basi di dati. In particolare, si descrive l'uso di LEX, un generatore automatico di analizzatori lessicali e di YACC, un generatore automatico di analizzatori sintatttici, disponibili in ambiente UNIX.

Segue a conclusione, una bibliografia fondamentale commentata, sui "sistemi esperti", al cui tema NOTE DI SOFTWARE intende dedicare ampio spazio nei prossimi numeri, e una recensione del recente libro "Software Engineering - Analysis and Verification", di T. G. Lewis.

La redazione

DISCUSSIONI:

PROTOTIPIZZAZIONE: SPUNTI ED ESPERIENZE DA MILANO

raccolte da
Gianni Degli Antoni
Istituto di Cibernetica, Università di Milano

Introduzione

Lo sviluppo di sistemi operativi dotati di buoni strumenti di programmazione e di librerie ben fornite accanto alla riduzione dei costi degli elaboratori e dei terminali che rendono possibile la programmazione interattiva, suggeriscono la ricerca di nuove forme di programmazione. La comparsa di un vasto mercato di software suggerisce di ricercare anche nella qualità la ragione del successo di alcuni prodotti. Tutto ciò accanto alla evidenza che talvolta il software di qualità è spesso software di autore, ovvero è realizzato da poche persone di cui si conosce il nome e di cui è eventualmente nota la competenza, hanno suggerito di considerare come possibile metodologia di programmazione il modo impiegato in ambiti universitari dove essenzialmente poche persone con una limitata divisione del lavoro cooperano alla realizzazione di prodotti spesso rifacendo completamente parti piuttosto che tentare di migliorarle allorchè vengono valutate come inadeguate. A tali metodologie viene spesso assegnato il nome di metodologie di propotipizzazione.
Anche presso l'Università degli Studi di Milano (Istituto di Cibernetica) è in fase di sperimentazione e messa a punto una metodologia di programmazione che si basa sui presupposti indicati.
Lo scopo di questa nota è di inquadrare in un unico lavoro le esperienze di molte persone cercando di interpretarne il significato costruendo un modello che possibilmente consenta di analizzare le fasi della metodologia ed eventualmente introdurne delle varianti. Questa nota non intende proporre il modello presentato quale unica forma di metodologia di propotipizzazione: viceversa ritiene indispensabile che alla metodologia vengano apportate modifiche per meglio adattarla alle esigenze dell'ambiente produttivo in cui si opera.

Le ipotesi organizzative del Modello

La metodologia che discuteremo si basa sulle seguenti ipotesi, tipiche degli ambienti di ricerca:

a) L'estensore dei programmi è anche il responsabile del progetto che tuttavia deve rispondere dello stato del progetto a colleghi o superiori: lo indicheremo con il termine di progettista.
b) Il progettista dispone di una adeguata stazione di lavoro dotata di strumenti di programmazione con un buon livello di interattività.
c) Il progettista dispone di buone librerie di programmi per problemi molto comuni.
d) La scadenza del progetto non è fissata esplicitamente, viene tuttavia prevista dal superiore del progettista e dai suoi colleghi.
e) È disponibile un documento iniziale che precisa in modo non necessariamente formale le specifiche del progetto da realizzare.
f) Alcuni colleghi a cui il progettista deve rispondere hanno una forte competenza nei problemi di programmazione e di architettura di sistemi.
g) Vengono coinvolti i futuri utenti durante lo sviluppo.

Il modello organizzativo.

A partire dalle specifiche iniziali il progettista procede ben presto alla implementazione del sistema da realizzare. Successivamente durante opportune SESSIONI DI PROTOTIPIZZAZIONE il progettista presenta lo stato del progetto direttamente sul terminale e vengono effettuate discussioni. Alle sessioni di prototipizzazione partecipano superiori, colleghi, utenti ed eventualmente esperti qualora colleghi e superiori non abbiano alcune fra le competenze necessarie per la evoluzione del progetto. Eventuali SUGGERIMENTI forniti durante le sessioni di prototipizzazione vengono immediatamente registrati dal progettista su un opportuno file con la indicazione di chi ha fornito il suggerimento. Il progettista, sulla base dei suggerimenti da lui raccolti durante le sessioni di prototipizzazione e sulla base della congruenza di tali suggerimenti con gli obiettivi di progetto, deciderà opportune AZIONI DI PROTOTIPIZZAZIONE. Tali azioni potranno riguardare: modifiche o introduzione di nuove funzionalità, modifiche di architettura sia a livello concettuale che a livello implementativo, modifiche alla documentazione, valutazione di prestazioni, attività di testing...
Il progettista raccoglierà le AP in un opportuno file tenendo un minimo di traccia delle ragioni delle scel-

te soprattutto allorchè le stesse sono determinate come conseguenza di già registrati suggerimenti.
Il progettista segnalerà ai superiori ed utenti le azioni di prototipizzazione intraprese e fisserà la prossima sessione durante la quale presenterà il progetto assieme alle implementazioni attuate delle azioni di prototipizzazione.

Commenti sul modello organizzativo

Il modello organizzativo assunto è orientato ad ottenere un ambito creativo competitivo e partecipativo. Tale ambito è costituito dalle sessioni di prototipizzazione. La creatività è conseguenza della competizione indotta dalla presenza di partecipanti a vari livelli gerarchici e dalla consapevolezza che i suggerimenti potranno venire accolti. La disponibilità ad accogliere suggerimenti da parte del progettista è essenziale. Nella fase iniziale dello sviluppo della metodologia la natura innovativa della stessa sembra favorire tale disponibilità. Alcune esperienze realizzate sembrano supportare questa ultima affermazione. È d'altra parte assai chiaro che se il divario di competenze fra progettista e superiori o colleghi che forniscono suggerimenti è sufficientemente alto, allora le difficoltà segnalate dovrebbrero ridursi e forse compaiono difficoltà di segno opposto: ovvero difficoltà a fornire suggerimenti. L'autore non ha mai osservato questa ultima forma di comportamento.
La separazione fra suggerimenti e azioni di propotipizzazione è rivolta a favorire la crescita di competenza autonoma da parte del progettista ed a garantigli autonomia di scelta, e quindi effettiva responsabilità.

Le tecniche di programmazione impiegate

Durante una fase iniziale il progettista realizza il progetto impiegando il più possibile funzionalità di libreria o modifiche a programmi di cui può disporre affine di poter sperimentare al più presto un prototipo funzionante. Successivamente, sulla base di problemi di prestazioni, di architettura o semplicemente come conseguenza della volontà di realizzare software di proprietà, rifarà completamente le parti prese in prestito.
Questa tecnica viene spesso indicata con il termine di fast prototyping. Nello schema che stiamo mostrando il fast prototyping viene impiegato come tecnica di programmazione e riguarda solamente la vita iniziale del progetto. È importante osservare che la tecnica indicata pone fine alla così detta tecnica del "divide et impera". In effetti la costruzione di programmi procede in maniere del tutto differente a partire dalla lettura della documentazione e/o di programmi disponibili e da una conseguente modifica. La efficacia del fast prototyping è legata alla disponibilità di una buona documentazione in linea ed alla possibilità di ricercare programmi con tecniche di "algorithm retrieval", accanto a opportuni strumenti di programmazione.

Le tecniche di documentazione

La documentazione gioca un ruolo importante in tutte le metodologie. Nel caso del modello che stiamo presentando è facile prevedere quando affermiamo: poichè assumiamo una forte competenza di programmazione, la documentazione viene ridotta all'essenziale, ovvero alla descrizione sintetica ed accurata delle funzionalità da realizzare ed a brevissime presentazioni dei passi più oscuri dei programmi realizzati. Il tutto preceduto da una descrizione anche architetturale dell'intero sistema.
La documentazione è predisposta e modificata costantemente durante lo sviluppo il più possibile immediatamente prima o immediatamente dopo lo sviluppo di certi programmi. La eventuale disponibilità di strumenti interattivi in cui il progettista predispone nello stesso contesto documentazione e programmi da cui poi automaticamente attraverso tecniche di UNDOC vengono estratti i programmi, sembra poter risolvere il problema del rapporto fra testo dei programmi e la loro documentazione in un modo definitivo.

La crescita della competenza

Il modello di prototipizzazione ha in comune con altri la relativa semplicità di trasferimento della conoscenza e quindi la crescita della competenza facendo. La competenza che viene acquisita, facendo, riguarda tutti gli aspetti del progetto: il settore a cui il progetto si applica, la reazione degli utenti, le tecniche implementative, la architettura di sistemi, ecc.
La acquisizione sarà tanto più alta quanto più la sessione di prototipizzazione fornisce spunti per osservazioni risolutive di difficoltà che il progettista ha incontrato o quanto più vengono messi in evidenza aspetti significativi del progetto.
È da notare che le capacità didattiche di chi partecipa alla sessione di prototipizzazione possono favorire la crescita delle competenze del progettista. Non occorre infine sottilineare che le capacità del progettista a tenere conto dei suggerimenti ed ad anticiparli costituirà un forte stimolo per la crescita della sua competenza o per la qualità del progetto.

Il caso di sistemi interattivi

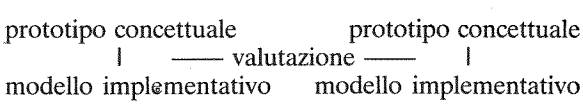
Lo sviluppo di sistemi interattivi di buona qualità può essere realizzato solo attraverso tecniche di prototipizzazione. Tecniche che non permettono la sperimentazione di ciò che si intende fare hanno scarso valore in quanto è estremamente difficile produrre una buona documentazione e buone specifiche di sistemi iterattivi. Le tecniche di prototipizzazione sono inoltre essenziali qualora non si disponga di personale con sufficiente competenza. Infatti la ralizzazione di prototipi e la critica attuata nelle sessioni di prototipizzazione favoriscono la acquisizione di competenza.

Il problema centrale: quando rifare?

I suggerimenti e le conseguenti azioni tendono a modificare le ipotesi iniziali ed a rendere eventualmente complessa la introduzione di nuove funzionalità od a ridurre le prestazioni al di sotto dei limiti accettabili.
In entrambe queste condizioni una accurata analisi dele basi architetturali dovrà essere attuata ed una decisione dovrà scaturire: rifare parti o tutto. Naturalmente rifare tutto significherà modificare la architettura del sistema, modificare cioè la natura e la rappresentazione degli oggetti su cui si basa l'intero sistema, le operazioni di base e il rapporto utente sistema. La acquisizione di competenza sulle condizioni per il rifacimento è uno di più importanti aspetti della metodologia di prototipizzazione proposta. La esistenza di documentazione sulla vita del progetto rende possibile la presenza di figure che effettuino una attività di supervisione dell'intero progetto e che siano in grado di prevedere difficoltà durante il suo svolgimento.

Il ciclo di vita della prototipizzazione

In questo paragrafo cercheremo di fornire una breve visione della metodologia di prototipizzazione così come si delinea nelle parti precedenti del presente lavoro.
Per questo è sufficiente osservare che alla base di qualsiasi attività di implementazione vi è una visione concettuale astratta del sistema da implementare, anche nei casi in cui non siano stati fatti sforzi per mettere in evidenza un tale prototipo concettuale. È a partire da questo che si costruisce una implementazione, ovvero un particolare modello del prototipo concettuale rappresentato attraverso opportune scelte di base ben rappresentabili con strutture computazionali.
Il ciclo procede dunque così:



Strumenti per la prototipizzazione

Sono di due tipi: per la gestione delle sessioni di prototipizzazione e per lo sviluppo.
Un esame della metodologia presentata indica che molta iniziativa viene lasciata al progettista su come e quando attuare decisioni: ciò comporta la adozione di strumenti che si facciano carico del coordinamento. Poichè il coordinamento fra parti distinte di attività richiede molta intelligenza, ciò può essere effettuato solamente attraverso la adozione di strumenti interattivi: per scrivere programmi, per effettuare test, per documentare, per effettuare valutazioni di prestazione, per attuare modifiche, ...

Su alcune limitazioni

Abbiamo taciuto su due aspetti certamente ritenuti fondamentali dal lettore: il tempo di sviluppo e la dimensione del progetto.
Sul primo possiamo esprimere solo opinioni, sul secondo uno abbiamo ancora effettuato nè analisi nè esperienze particolarmente significative. Tuttavia possiamo affermare che la partecipazione di più persone che si coordinano informalmente ad uno stesso progetto senza divisione del lavoro predisposta, non richiede alterazioni alla metodologia presentata.

Le persone che hanno contribuito e che stanno operando, con proposte ed esperienze, per la messa a punto della metodologia qui presentata sono:

Gianluigi Castelli
Giorgio Castiglioni
Antonio Cicu
Marta Ferrari
Marco Maiocchi
Marco Meli
Roberto Polillo
Gianpaolo Rossi
Carla Simone
Luca Spampinato
Guido Torriani
Marco Villa

CONTRIBUTI TECNICI:

GLI STANDARDS DI DOCUMENTAZIONE TECNICA NELLA PRODUZIONE INDUSTRIALE DI SOFTWARE (1)

A. Cicu – Honeywell Information Systems Italia – Milano
C. Cucciati – Honeywell Information Systems Italia – Milano
M. Maiocchi – Istituto di Cibernetica – Università di Milano

1. IL SOFTWARE: EVOLUZIONE DEL CONCETTO DI PRODOTTO.

Prima di discutere i problemi di produzione del Software connessi con la documentazione, è opportuna una premessa su cosa si intenda per prodotto Software e su come si sia evoluto questo concetto con il consolidarsi dei metodi e degli strumenti per produrre Software.

Il Software era visto, nei primi anni di sviluppo dei calcolatori, solamente come l'insieme dei programmi scritti per controllarne il funzionamento e risolvere con essi certi problemi applicativi.

L'unica documentazione di corredo era il manuale dell'utilizzatore.

È stata l'analisi dei problemi della qualità del Software e lo studio dei mezzi e dei metodi *per migliorare la qualità* da un lato e per *ridurre i costi di sviluppo e di manutenzione* dall'altro che ha portato a ulteriori raffinamenti del concetto di prodotto Software, e a capire (e formalizzare) la struttura del processo di produzione.

Sono comparse quindi in letteratura definizioni del prodotto Software che includevano esplicitamente, come parte integrante del prodotto, anche la documentazione interna di ingegneria.

Per esempio:

"Il Software di sistema è un insieme organico, integrato e non scindibile di programmi per calcolatore da una parte, e di manuali di utilizzazione dall'altra che ne stabiliscono le regole d'uso, e ancora di documentazione interna che ne consente la manutenzione e la gestione" (CICU 70).

"We will define Software to include not only computer programs, but also the associated documentation required to develop, operate, and maintain the programs... to emphasize the necessity of considering the generation of timely documentation as an integral portion of the Software development process" (BOEHM 76).

Una ampia serie di studi, che hanno portato all'attuale diffuso sviluppo di ricerche e sperimentazioni sull'Ingegneria del Software, ha fatto comprendere che qualsiasi razionalizzazione e automazione del processo di produzione del Software non può limitarsi a cosiderare il prodotto Software come un'entità statica (i programmi e i relativi documenti), ma anche che di detto prodotto devono essere tenuti assai presenti gli aspetti dinamici della sua evoluzione durante il suo ciclo di vita e di interazione con gli ambienti dove esso viene prodotto, insegnato, usato, riparato, esteso nelle sue funzionalità. Conseguentemente si è chiarito che i criteri guida che devono ispirare le attività di produzione non sono solamente quelli di struttura del prodotto, ma soprattutto quelli connessi con le funzioni, e gli scopi intrisechi del prodotto Software stesso (cioè ciò che interessa all'utente), e quelli connessi con la produttività e i costi di produzione, manutenzione e uso.

Su questa base si è arrivati a estendere in modo significativo il concetto di prodotto Software.

In un lavoro di Ross (ROSS 76) si afferma che soltanto avendo presente un concetto allargato di "Software" noi possiamo capire come "gli obiettivi rispetto all'esigenze dell'utente siano in definitiva tradotti, incarnati in sistemi di calcolo funzionanti". Dice Ross: "Software embodies *every* aspect of actually using computers and communication equipments in an application area... Therefore Software is far more than just programs. Sofware includes all of the planning, design, operating procedures, language conventions..."

Negli standards del ciclo di produzione Software HISI (HISI 79) (METOD 28) è stata adottata una definizione del prodotto Software comprensiva non

(1) Questo lavoro è stato presentato al Convegno FAST/CNR organizzato nell'ambito del Sottoprogetto METOD (Progetto Finalizzato per l'Informatica del Consiglio Nazionale delle Ricerche) tenutosi a Milano il 27-28-29 settembre 1982.

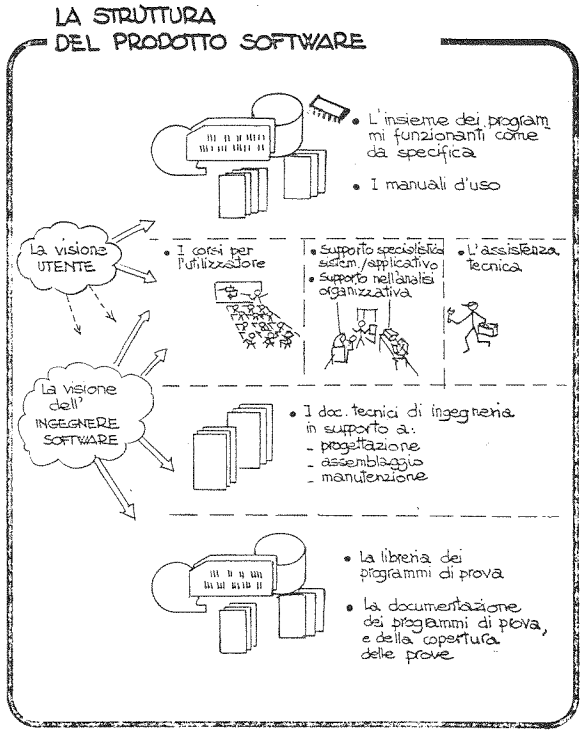


Fig. 1

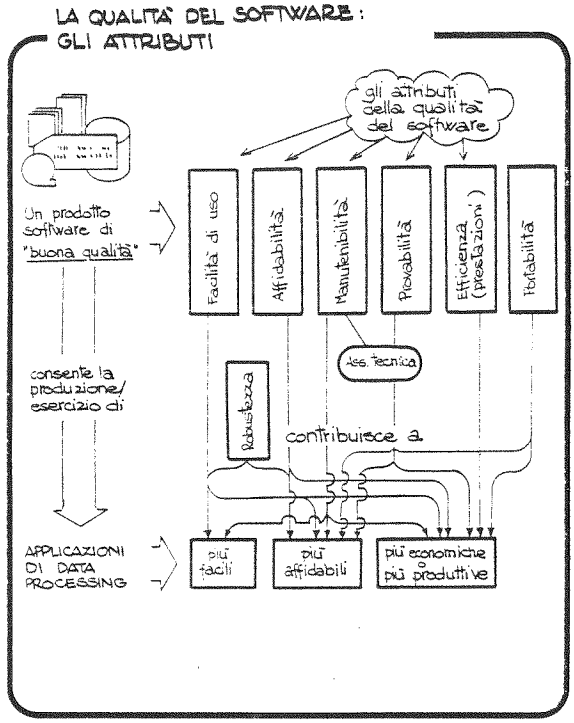


Fig. 2

solo agli aspetti di documentazione esterna ed interna, ma anche degli aspetti di *servizio* e di *supporto* connessi con l'uso del prodotto. Infatti oggi l'utente sempre di più chiede insieme con il pacco dei programmi anche la fornitura di servizi di assistenza applicativa, tecnico/sistemistica, e di formazione. (Si veda la Fig. 1, – da METOD 18 – sulla *struttura* di un prodotto Software).

H.D. Mills, in un recente lavoro sui principi dell'Ingegneria del Software (MILL 80), esprime concetti analoghi: "... the term Software means... logical doctrine for harmonious cooperation of a system of people and machines, ... where the agents of action are people and machines, with the blueprints for their action supplied by software. A human procedure is as important to the system as a machine procedure... that is, Software is typically a set of logical blueprints for the operation and use of a multi/distributed processing system by an organization of people in its natural evolution over time."

Inoltre, sempre negli stessi standards HISI, si è posta l'enfasi su quali sono gli *scopi* ultimativi per cui l'utente investe nel Software: che sono riassunti nel rendere più facili, più affidabili, e più economiche o più produttive le sue applicazioni di elaborazione di informazioni. La stessa *qualità* di questo prodotto è giudicata dall'utente attraverso attributi tipici (CICU 70, CICU 81/2) che sono strettamente correlati agli scopi che l'utente persegue con il Software (si veda la Fig. 2) da (HISI 79), (METOD 28), (CICU 81/2).

Questa introduzione sul concetto di prodotto Software, cioè sulla sua *struttura* e sui suoi *scopi*, è volutamente propedeutica al tema da trattare (la documentazione del Software): la *struttura* infatti ci fornisce indicazioni sugli *oggetti* che l'attività produttiva deve creare, e gli *scopi* del prodotto ci forniscono i *criteri ispiratori* dei requisiti di qualità e di costo a cui tali oggetti devono soddisfare.

2. LA DOCUMENTAZIONE DEL PRODOTTO SOFTWARE: PROBLEMI E EVOLUZIONE STORICA

2.1 La carenza iniziale e sue cause

La documentazione dei programmi è stata fin dall'inizio dello sviluppo dell'arte di fare Sotware uno degli aspetti più trascurati, anche dalle persone professionalmente più preparate.

La tradizione di scarsa o nulla attenzione devoluta alla documentazione pesa ancora oggi: è frequente trovare interi gruppi di analisti/programmatori anche esperti che non documentano i programmi e che razionalizzano questo comportamento affermando che la documentazione è poco utile e accresce in misura inaccettabile i tempi e i costi di progetto.

Ciò che è avvenuto nella documentazione del Software è esattamente l'opposto di ciò che si può verificare nella progettazione e produzione di qualsiasi

altro prodotto industriale. Si pensi ad una casa, ad una automobile, ad una macchina qualsiasi, i quali tra l'altro sono tutti oggetti fisicamente visibili, palpabili nelle loro forme e dimensioni (in altre parole essi sono oggetti "Hardware"). Per questi oggetti l'architetto e l'ingegnere sanno bene che senza disegni completi dell'*insieme* e *dettagliati per ogni parte componente,* e senza documentazione completa delle *dimensioni e forme* di ogni componente e della coerenza degli aspetti di interfaccia/corrispondenza tra le parti contigue, i detti oggetti non verrebbero mai costruiti. Non solo, architetti e ingegneri sono abituati a specificare in dettaglio la *sequenza* di costruzione e di montaggio delle varie parti, nonchè le norme di collaudo parziale e finale. Infine, proprio basandosi sulla *lista componenti* e sulla conoscenza a priori delle operazioni di montaggio, ingegneri e architetti sono sempre stati capaci di stimare con buona approssimazione i *costi a preventivo,* almeno per gli aspetti di produzione.

Lo "Hardware" stesso dei calcolatori, quando si è dovuto affrontarne la produzione in serie, è stato sempre ben documentato, fin dall'inizio.

Si può affermare che nel caso dei prodotto industriali di tipo "Hardware" l'obbligo di una buona documentazione è sempre derivato dagli *alti costi* degli investimenti in macchinari *per la produzione in serie* e *dal costo intriseco degli oggetti scartati* perchè riusciti male o sbagliati. I costi delle modifiche o delle correzioni in questi casi sono tali da costringere ingegneri, architetti e direttori di produzione a spingere al massimo le verifiche preliminari di coerenza tra le parti, appoggiandosi a documenti molto accurati e completi.

Nel caso ancora dei beni "Hardware" la necessità di costruire dei *prototipi* prima di avviare una produzione ha sempre aiutato a perfezionare e precisare i documenti comunque necessari, e nei casi in cui non è possibile costruire dei prototipi (case, ponti, navi, ecc.) l'ingegneria è sempre ricorsa alla costruzione preliminare di *modelli* per verificare preliminarmente il maggior numero di prestazioni statiche e dinamiche.

Nel prodotto Software tutto ciò non è avvenuto (se non parzialmente in alcuni casi): nonostante che questo prodotto avesse un motivo in più rispetto allo "Hardware" per essere documentato, perchè il Software nel suo stato di prodotto finito non è visibile nè palpabile.

A differenza di altri prodotti industriali, il Software è *visibile e tangibile*, per sua natura, *solo attraverso la sua documentazione e il suo uso.*

Il Software è un prodotto industriale di natura particolare (HISI 79) (METOD 28, parte I): il *Software è informazione*, un tipo particolare di informazione inventato per comandare e controllare il funzionamento dei calcolatori (meccanici, elettromeccanici, elettronici).

Siffatta informazione per sua natura esiste in più stati (uno stato in forma simbolica direttamente comprensibile alla mente e all'occhio umano, e un secondo stato in cui l'informazione stessa è codificata in sequenze di comandi binari 0/1 comprensibili alla macchina e con assai maggiore difficoltà alla mente umana).

Allo stato di prodotto finito questa informazione è – si è detto – una sequenza di comandi binari, registrata su schede o nastri perforati (all'inizio della storia dei calcolatori), o su supporti magnetici (in origine i nastri, poi, come oggi nella maggioranza dei casi, su dischi magnetici, o dischette flessibili, o nastri-cassette), o addirittura in modo stabile in memorie permanenti della macchina "Hardware" stessa.

La conseguenza è che il Software, nel suo stato di prodotto finito, *è un prodotto invisibile* (ibidem).

Perchè dunque è stato possibile non solo produrre Software, ma anche sviluppare e far crescere questa industria, pur non prestando la giusta attenzione iniziale alla documentazione?

I motivi possono riassumersi nei seguenti:

a – il prodotto Software è correggibile e modificabile con costi assai più bassi di una modifica Hardware.

b – il costo basso della fase "produzione di serie" del Software

c – il tipo di professionisti e le piccole dimensioni dei gruppi, impegnati nei primi anni di sviluppo dell'arte del Software

d – la novità di questa arte e il suo tasso di sviluppo

a) il prodotto Software è correggibile e modificabile con costi assai più bassi delle modifiche Hardware.

Mentre la correzione o modifica dell'Hardware richiede una modifica dei disegni, dei layout di una piastra, di uno o più componenti e spesso una riprogrammazione della linea di produzione e delle operazioni di collaudo, la equivalente operazione su un programma è fattibile sostituendo le istruzioni errate con quelle corrette in modo generalmente rapido e automatico (correzioni ai programmi sorgenti e ricompilazioni, o addirittura interventi sul codice oggetto con l'applicazione delle cosiddette "patches" (pezze).

Soprattutto nei primi anni di sviluppo, quando i sistemi Software erano fatti di programmi relativamente piccoli, con pochi moduli interfacciantisi, la facilità e rapidità di intervento erano relativamente elevate (oggi, con l'aumento intervenuto delle dimensioni dei sistemi operativi, sulle correzioni pesano in modo accresciuto soprattutto i tempi di "debugging" e di ricollaudo, tuttavia, in termini relativi allo Hardware, il discorso si può ritenere ancora valido) e ciò portava il progettista a trascurare o a rinviare ad un secondo tempo le attività documentative: ciò era favorito dal fatto che la correzione era effettuata quasi sempre dalle persone competenti che avevano scritto il programma, che non avevano perciò esigenza di apprendere da altri le informazioni sulla struttura del programma stesso.

b) Il costo basso della fase "produzione di serie" del Software.

Lo Hardware presenta due fasi di sviluppo ben distinte: la prima è la progettazione e costruzione dei prototipi di ingegneria, seguita dalla fase di produzione di serie.

La seconda fase presenta costi e tempi non trascurabili: il decrescere di detti costi e tempi dipende certo dai volumi, ma anche in modo determinante dall'accuratezza della documentazione e dei collaudi effettuati sul prototipo.

Il Software, almeno come è tradizionalmente inteso, presenta una fase di "produzione di serie" assai semplice: essendo il Software informazione, la sua produzione di serie coincide con la duplicazione dei supporti sui quali sono registrati i programmi, e con la riproduzione della documentazione. Ambedue i processi sono oggi automatici e relativamente rapidi. Ciò porta a poter riprodurre facilmente un Software corretto, e soprattutto a poterlo riprodurre anche se le modifiche non sono documentate.

Inoltre, almeno finchè le dimensioni dei progetti Sofware sono rimaste piccole, l'analista/programmatore doveva produrre il suo programma come un prodotto finito; progetto e produzione perciò quasi coincidevano, e il progettista non aveva perciò il problema di dover comunicare ad altri come fare e che cosa fare.

L'esigenza di una buona documentazione è nata con l'aumento delle dimensioni dei prodotti, con l'allungarsi della loro vita e con l'emergere di problemi significativi di integrazione e manutenzione.

c) Il tipo di professionisti e le piccole dimesioni dei gruppi impegnati nei primi anni di sviluppo dell'arte del Software.

I primi professionisti del Software avevano in generale una formazione di tipo matematico o fisico; i programmatori delle prime applicazioni di tipo amministrativo avevano una formazione di tipo finanziario o amministrativo.

In nessuno di questi casi nè la formazione scolastica nè l'esperienza di lavoro dava sufficiente importanza e ruolo alla documentazione, come invece era già tradizione per gli ingegneri.

Inoltre, come già accennato, le dimensioni ridotte dei primi gruppi di programmatori favorivano la comunicazione diretta tra le persone addette a scapito ancora della documentazione.

Il risultato è stato spesso di una scarsa o nulla visibilità dei lavori e dei problemi Sofware da parte del management, e i gruppi di programmatori erano visti come un mondo esoterico, impenetrabile e ingestibile sia presso i produttori di sistemi che presso i centri EDP: gli addetti stessi cominciarono a capire che senza documenti loro stessi non erano in grado di promuovere e difendere il loro lavoro.

d) La novità di questa arte e il suo tasso di sviluppo.

La novità dell'arte del Software ha fatto sì che all'inizio la creatività sia stata dedicata più ai problemi che con i programmi si volevano risolvere che non ai metodi.

L'obiettivo costante in tutti gli sviluppi del Software è stato un continuo accrescimento della "facilità di uso" in tutti i settori applicativi (CICU 81/2): e ciò ha portato naturalmente gli addetti a concentrare gli sforzi nell'inventare Software sempre con nuove funzionalità e ad aumentare la complessità, a scapito, inizialmente, di una sistemazione dei metodi e dei documenti.

2.2 Gli sviluppi della documentazione nel Software.

Si è già inizialmente (par. 1) accennato che sono stati gli sforzi intrapresi per dominare i problemi di qualità del Software, nonchè per ridurre i tempi e costi sia di sviluppo che di manutenzione che hanno portato alla diffusa consapevolezza attuale sul ruolo della documentazione (sintesi in Fig. 3).

La *centralità*, il *ruolo chiave* della documentazione nel processo produttivo del Software sono stati sempre sottolineati fin dai primi studi di ingegneria del Software (LECH 67, NATO 68, WALS 69, CICU 70, KATZ 71). Negli atti della conferenza del 1968 organizzata dal NATO Science Committee sul Software Engineering (NATO 68) si può vedere che la documentazione era l'aspetto più ricorrente nella di-

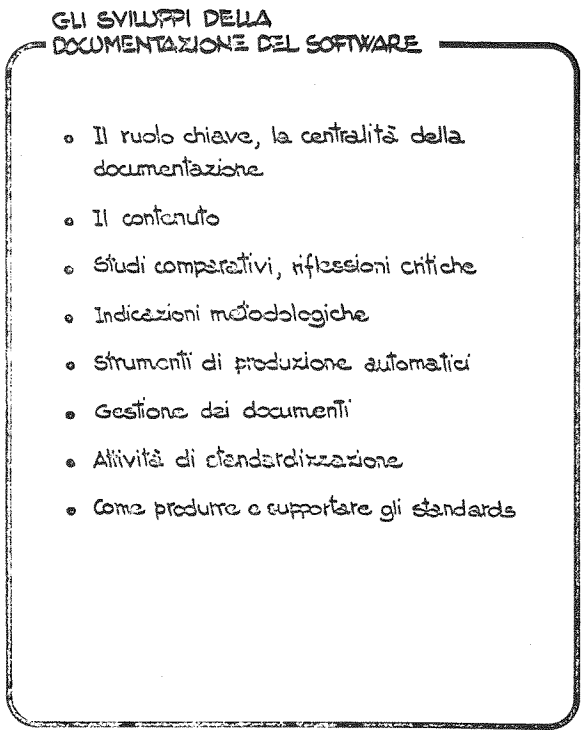


Fig. 3 - Lo schema di figura sintetizza i settori ove si sono avuti contributi significativi, nell'ultimo decennio, nel sistematizzare l'arte di documentare programmi.

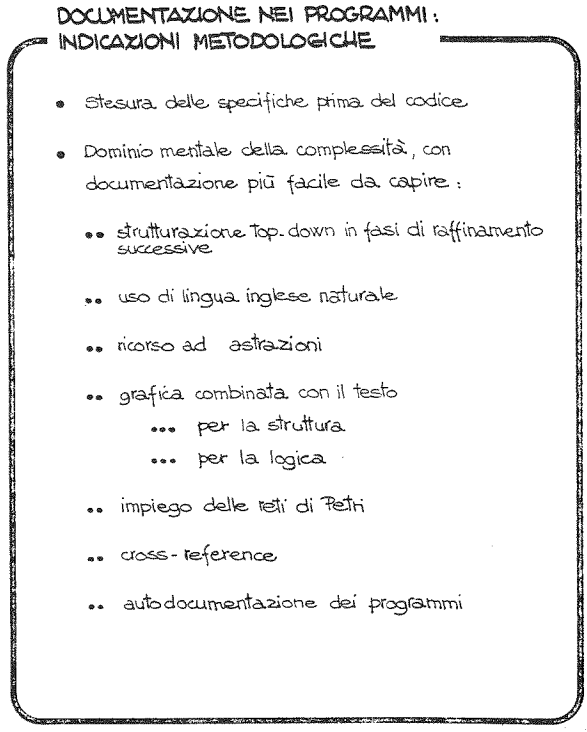


Fig. 4 - Sintesi delle molteplici indicazioni metodologiche che sono state suggerite e sperimentate a favore della completezza del disegno e di una più facile comprensione dei programmi.

scussione (vedi il Subject Index finale: in particolare contribuirono Dijkstra, Naur, Selig, Fraser, Smith, Harr).

In uno studio di modelli formali del processo di sviluppo (BELA 76), la documentazione è citata come una delle assunzioni alla base del modello, e come parte integrante degli oggetti prodotti nel processo, le cui deficienze sono dichiarate essere causate da dificienze di documentazione.
Il contenuto dei documenti è stato curato in modo sempre più accurato.
Il ruolo della documentazione è pure completamente riconosciuto nei requisiti per gli Ada Programming Support Environments (DOD 80), ove, nel delineare le caratteristiche fondamentali di un sistema di documentazione per il software, si delinea esplicitamente che una parte sostanziale dello sforzo di sviluppo è devoluto alla documentazione.

Recentemente addirittura il codice è visto secondario rispetto alla documentazione che è considerata il vero prodotto (NEWM 82).

La documentazione del Sofware è più in generale degli interi sistemi informativi è stata oggetto di *studi comparativi* e di *riflessioni critiche* sull'influenza della documentazione nel processo produttivo. Si vedano la rassegna sull'evoluzione dei metodi di analisi dei sistemi in corrispondenza dell'evolvere dei sistemi stessi (COUG 73), le analisi sulla influenza della documentazione sulla produttività e sulla qualità (DEGL 76, BELF 76, DEGL 77/1), e le considerazioni sulla correlazione tra linguaggi di specifica, metodi di disegno e la facilità di uso, comprensione e traduzione della documentazione risultante (JONE 81).
Rilevante è l'insieme delle *indicazioni medologiche* sviluppate nell'ultima decade (sintesi in Fig. 4):

a) L'insistenza sulla necessità di *stendere le specifiche prima di produrre il codice* è generale, si vedano in particolare (LECH 67), (Naur e Dijkstra in NATO 68), (KATZ 71), (BROW 74), (YOHE 74), (BOEH 76), (HIEM 75), (KATZ 76).

Indicazioni precise e/o guide sul *contenuto* di informazioni dei vari documenti sono state pubblicate, con dettaglio e cura crescente nel tempo, in vari lavori, nei quali i raffinamenti successivi hanno reso il contenuto di un documento sempre più aderente alle problematiche trattate nella fase del ciclo di produzione in cui si prepara il documento stesso (LECH 67, Selig in NATO 68, Naur in NATO 68, WALS 69, CICU 70, TRAU 73, METZ 73, NAUR 74, YOHE 74, BROW 74, GOOS 75, KATZ 76, BOEH 76, NEWM 76, TAUS 77, DONA 78, TAUS 79, HISI 79, IEEE 81/1, METO 28, METO 31).

b) l'esigenza di rendere più facile il *dominio mentale della complessità, con il supporto di una documentazione più facilmente comprensibile* ha originato sia studi specifici (BROO 78, BROO 81) che contributi in varie direzioni:

b1) la *strutturazione top - down* delle descrizioni e delle specifiche stesse in *fasi di raffinamento successive* (DIJK 72, NAUR 74, BROW 74, BERR 75, MILL 75/1, KATZ 76 (il metodo HIPO), BUCK 78)

b2) l'uso della *lingua inglese naturale* per spiegare più facilmente il significato delle procedure (SAMM 66, PARN 72/1, MILL 75/1, HOBB 77, BALZ 78)

b3) il ricorso ad *astrazioni* (concetti espressi in modo narrativo o grafico), affiancate da spiegazioni dettagliate (il metodo Pagina-Si, Pagina-No (PSPN) (DEGL 78) (MASE 77), l'uso di metafore (DEGL 77/2)

b4) il ricorso all'uso di *strumenti grafici combinati con il testo* per rendere facilmente afferrabile alla mente la *struttura del programma* (STAY 76 e KATZ 76 con HIPO, JACK 75 con JACKSON, BETT 77 e MAIO 79 con PHOS, ROSS 78 e SOFT 76 con SADT, LANO 79 con le N^2-charts, GIMP 80 con gli structured flow-charts, BELA 80 con GREENPRINT), la *logica* di un componente (NASS 73, YODE 78, SHNE 77 con le carte di Nassi-Shneiderman), o i programmi interattivi tramite schemi di dialogo (CAPP 77). A questo riguardo va citato il lungo dibattito verificatosi a favore e contro l'uso dei flow-charts, e conclusosi con la constatazione che di fatto i flow-charts sono andati in disuso, e che ancora non si è scelto in modo predominante nessuno strumento tra quelli citati come il migliore per documentare, in modo grafico e ad alto livello, sia la *struttura* che la *logica,* anche se ne è confermata l'esigenza (ANS 79). Un accurato esperimento è stato condotto con programmatori per valutare gli effetti di diverse simbologie e diverse collocazioni spaziali dell'informazione sulla comprensione delle specifiche (SHEP 81).

b5) l'impiego delle *reti di Petri* nella descrizione dei sistemi e dei programmi (CAST 79, NDSW 10/11, CICU 81/1)

b6) l'importanza dei documenti (e relativi Tools) di *cross - reference* sia per i programmi che per i documenti stessi (GOOS 75)

b7) *l'autodocumentazione dei programmi* è stata considerata un passo decisivo verso una migliore manutenibilità, leggibilità, e per rendere più facile il debugging e le verifiche di correttezza, attraverso:

- la sottolineatura dell'importanza dei *commenti al codice* (Harr in NATO 68, GOOS 75)

- l'adozione di *standards di commentazione* del codice strutturati, a blocchi successivi di commenti, ciascuno contiguo al blocco di codice da descrivere e contenente le specifiche di Input/Process/Output di detto blocco (LANZ 77, GOBB 77/2, GOBB 78, CHIA 80), e con enfasi sulle *motivazioni* all'origine di ogni funzionalità (LANZ 77).
  Lo stesso Parnas, pur favorevole a nomi di variabili non evocativi del significato semantico, ha giudicato fondamentale il comunicare l'interpretazione voluta delle funzionalità di ogni modulo tramite descrizioni in linguaggio naturale (PARN 72/1).

- l'impiego di *asserzioni* opportunamente collocate nel corpo del codice del programma (MILL 70, MILL 75/2, STUC 75, CHOW 76, DEGL 76, MATU 76, LANZ 77, GOBB 77/1)

- l'uso di *strumenti* di produzione di documentazione integrati con la tecnica di autodocumentazione (CHIA 80) (vedi anche successivamente i riferimenti agli strumenti).

La *documentazione on-line* di interi sistemi (tipici gli esempi di MULTICS e UNIX con i testi di "help") ha indicato l'inizio di un nuovo approccio verso una maggiore produttività e una più facile apprendibilità.

Sempre in direzione di una maggiore *produttività e qualità nella stesura e manutenzione* di specifiche e di documenti sono da citare i continui progressi sugli *strumenti* di *creazione e trattamento* automatico *della parola, dei testi* e *della grafica.*

Si vedano (AIEL 80), (TEXT 81) e (TEXT 83) per una rassegna sugli strumenti di vario tipo disponibili.

Si deve notare che detti strumenti, concepiti normalmente per produrre documenti di qualità di qualsiasi tipo, sono stati ovviamente usati anche per produrre i documenti di specifica dei prodotti Sofware: ma nella quasi totalità dei casi le applicazioni al Software non hanno tenuto conto di alcune esigenze tipiche dei documenti di un prodotto Software.

I primi esempi di strumenti documentativi che hanno cercato di integrarsi nel processo produttivo del Software per facilitarlo sono stati gli "autoflowcharters"

(produttori automatici di flowcharts), i quali tuttavia hanno avuto il successo relativo connesso con la sorte dei flowcharts (considerati da tutti come troppo dipendenti dall'implementazione).

Più recentemente uno strumento di Orr (STRUCTURE, vedi (ORR 81)) è stato concepito per produrre e mantenere automaticamente i documenti di struttura di un programma (diagrammi di Warnier/ ORR, vedi ORR 77), e un secondo strumento (DOCUMENTOR, vedi CHIA 80) è stato realizzato, che consente la realizzazione e manutenzione delle specifiche di disegno in un modo molto integrato con lo sviluppo e la manutenzione dei programmi. In tale caso la documentazione dei programmi è creata sotto forma di blocchi di commenti associati ad ogni modulo, con vantaggi significativi sulla manutenzione del programma e del testo, e contemporaneamente è estraibile dal sorgente, anche per progetti composti da molti programmi, per produrre speficiche di progetto compatte e che si spingono ad un livello specificabile dall'utente sui dettagli della gerarchia top-down di un prodotto, consentendo perciò diprodurre facilmente descrizioni sintetiche ad alto livello anche di programmi grandi e rimandando all'esame del listato completo del programma per i dettagli minimi.
Un altro esempio di integrazione della documentazione con il codice del programma stesso è GREENPRINT (BELA 80). Questo strumento consente di stampare a fianco dei blocchi di programma uno schema a blocchi che visualizza in moso assai efficace la struttura dei blocchi stessi.
Un ulteriore esempio di documentazione integrata sul contesto è TESTDOC (CERI 81), che consente di creare e controllare la documentazione dei programmi di prova in un modo molto integrato e coordinato con le specifiche.

Si veda anche il lavoro (JONE 81) per le ricerche in corso sui linguaggi grafici (graphics design languages).

Non sono mancati anche contributi sugli *aspetti organizzativi di gestione della documentazione;* si tratta di semplici suggerimenti, la adozione dei quali tuttavia è di altissimo beneficio nella gestione di un progetto e nella rapidità di accesso ai documenti:

a) Una buona gestione dei documenti richiede anzitutto una *organizzazione ad albero dei documenti stessi, sia all'interno di un sistema che all'interno di un progetto (GOOS 75), e un meccanismo efficiente di consultazione (si veda un esempio in (LECL 82) e reperimento secondo gli argomenti).*
b) *Il secondo fondametale supporto di una buona gestione è il raccoglitore di progetto,* un semplice raccoglitore ad anelli suddiviso con separatori di cartoncino a linguetta sporgente, nel quale si conservano *tutti* i documenti di progetto nelle loro successive versioni, *tutti* gli appunti e i risultati delle successive versioni, *tutti* gli appunti e i risultati delle revisioni, *tutti* i piani, eccetera (si veda Opler in (NATO 68), e soprattutto (INGR 78) per una esposizione dei benefici dell'uso dell'Unit Development Folder, alias raccoglitore di progetto).

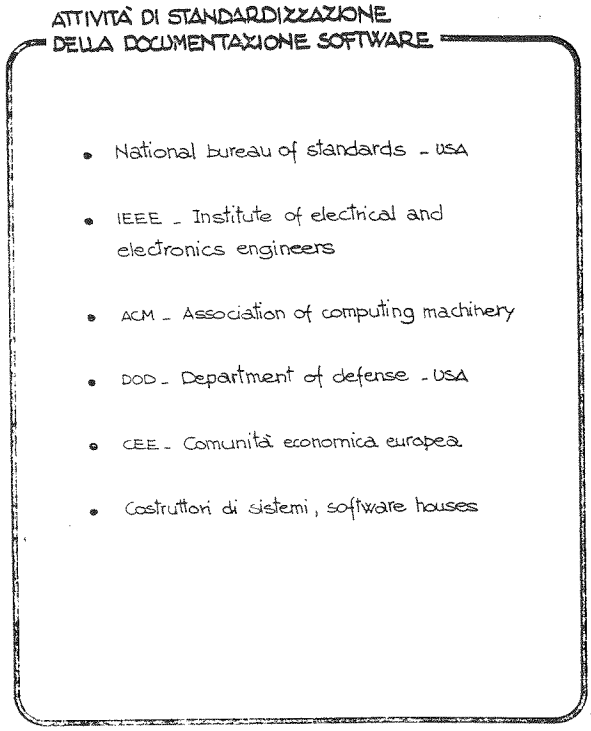


Fig. 5

Attività di standardizzazione e di studio sono espressamente *dedicate alla documentazione del Software* (sintesi in Fig. 5):

- *Il National Bureau of Standards* negli Stati Uniti ha da tempo emesso un suo insieme standards (FIPS standards: vedi FIPS 76, FIPS 79) attraverso il lavoro del Federal Information Processing Standard Task Group 14 e ha in corso una revisione del FIPS PUB 38 dopo i primi 4 anni di uso (KURI 81)

- l'*IEEE* (Institut of Electrical and Electronics Engineers) ha istituito un sottocomitato articolantesi in più task groups (BATE 81) per la standardizzazione della documentazione delle varie fasi dello sviluppo (un esempio è il progetto P 730 sul Software Quality Assurance Standard (IEEE 81/ 1), (BUCK 79), (BUCK 81)). Attualmente il medesimo sottocomitato sta lavorando per emettere nuovi standards sui seguenti argomenti: Configuration Management, Specifiche dei Requisiti, Documentazione delle prove e Terminologia (BUCK 81) (BATE 81)

- l'*ACM* (Association for Computing Machinery) ha istituito uno Special Interest Group for System Documentation (SIGDOC), che diffonde i contributi e le discussioni dei membri attraverso la rivista Asterisk

- *il Department of Defense* (DoD) americano è stato molto attivo in questo settore, avendo emesso già da tempo standards sulla gestione dei progetti e la stesura della documentazione (MSTD 74), e raffinandoli successivamente con requisiti anche quantitativi sulle misure di affidabilità (MSTD 78 SOUT 78, MSTD 79, BOWE 79, COOP 81). Questi ultimi standards hanno costituito la base per un recente lavoro (SUKE 81) che dopo sufficiente sperimentazione e revisione dovrebbe portare all'emissione di un manuale di guida, di uso assai diffuso, per le misure di affidabilità del Software.

- la *Comunità Economica Europea,* attraverso il Technical Comitee TC 7 dello European Workshop on Industrial Computer Systems, sta supportando un lavoro di standardizzazione, che riguarda tutti gli aspetti dell'Ingegneria del Software, documentazione compresa, con l'obiettivo di garantire con l'aiuto del calcolatore la sicurezza (Safety) di vari tipi di impianti (dai nucleari ai ferroviari) (BOLO 81)

- è recente il contributo (BROC 81) che ha trattato i problemi connessi con *l'insegnamento* dell'arte della documentazione, e che offre anche una rassegna sugli standards di supporto disponibili

- due *iniziative recenti* testimoniano l'interesse crescente sul tema della documentazione

  — l'International Conference on System Documentation (gennaio 1982) organizzata dagli Speciali Interest Groups dell'ACM sulla Systems Documentation (SIGDOC) e sull'Office Automation (SIGOA)

  — il FIPS Software Documentation Workshop organizzato dal National Bureau of Standards (marzo 1982)

  (per i dettagli su queste due iniziative si veda ASTERISK No. 5 del novembre 1981)

Contributi sono infine apparsi su *quali fattori* considerare e *quali passi* seguire nel *definire gli standards Software* che più si adattino a un dato tipo di progetto o a un dato ambiente, (PETE 81), *nel renderli facili da capire* e chiari sugli obiettivi essenziali (FOST 81), (GLAS 81), nel supportarne l'introduzione e l'uso iniziale e poi nel facilitarne l'evoluzione (WEDB 81).

3. LA DOCUMENTAZIONE E IL CICLO DI VITA DEL SOFTWARE

Tutte le trattazioni citate sul tema della documentazione del Software *collocano i documenti considerati nell'ambito di uno schema del processo di produzione,* ove ogni documento è uno degli oggetti prodotti in una determimata fase del processo stesso. Gli schemi del processo adottati variano lievemente da una trattazione all'altra: chi prevede una o due fasi in più della media, chi invece ne trascura alcune (o le assorbe in fasi più ampie). Sempre tuttavia le fasi considerate producono documenti e non solo programmi.

Questo ruolo centrale assunto dalla documentazione è confermato anche in recenti contributi relativi a prossime attività di valutazione di tools e metodologie (SIGS 81) (SEGA 81) (TRIP 81/1) (TRIP 81/2).

Come base della presentazione degli standards di documentazione in questo articolo si è assunto lo schema di ciclo di vita di un prodotto Sofware di Fig. 6 (HISI 79, METO 28, METO 31).

Per i dettagli sulle caratteristiche del ciclo di vita si faccia riferimento al documento (METO 21), oltre che a quelli citati.

Gli standards presentati sono quelli proposti nel documento (METO 31).

Lo schema di Fig. 6 si basa sui seguenti punti:

- il ciclo di vita del Software è un *processo ciclico* (METO 21). Un prodotto Software è sviluppato in versioni successive. In ogni versione si effettuano *modifiche* o *estensioni* per

  — correggere errori (modifiche)
  — adattarsi ai cambiamenti di ambiente (modifiche)
  — migliorare prestazioni (modifiche)
  — aggiungere nuove funzionalità (estensioni)

- il ciclo di vita del Software è un insieme di più attività suddividibili in 3 classi principali:

  — *produzione* vera e propria (include sia lo sviluppo che la manutenzione)
  — *controllo* del processo (piani e revisioni periodiche della qualità/costi/date)
  — fornitura/gestione dei *servizi di supporto*

- *produzione, controllo e gestione sono attività interdipendenti* e la gestione della revisione N del prodotto è sovrapposta al controllo e produzione della versione (N + 1), anche se, per comodità di rappresentazione, tale sovrapposizione è evidenziata in Fig. 6 solo in modo indiretto.

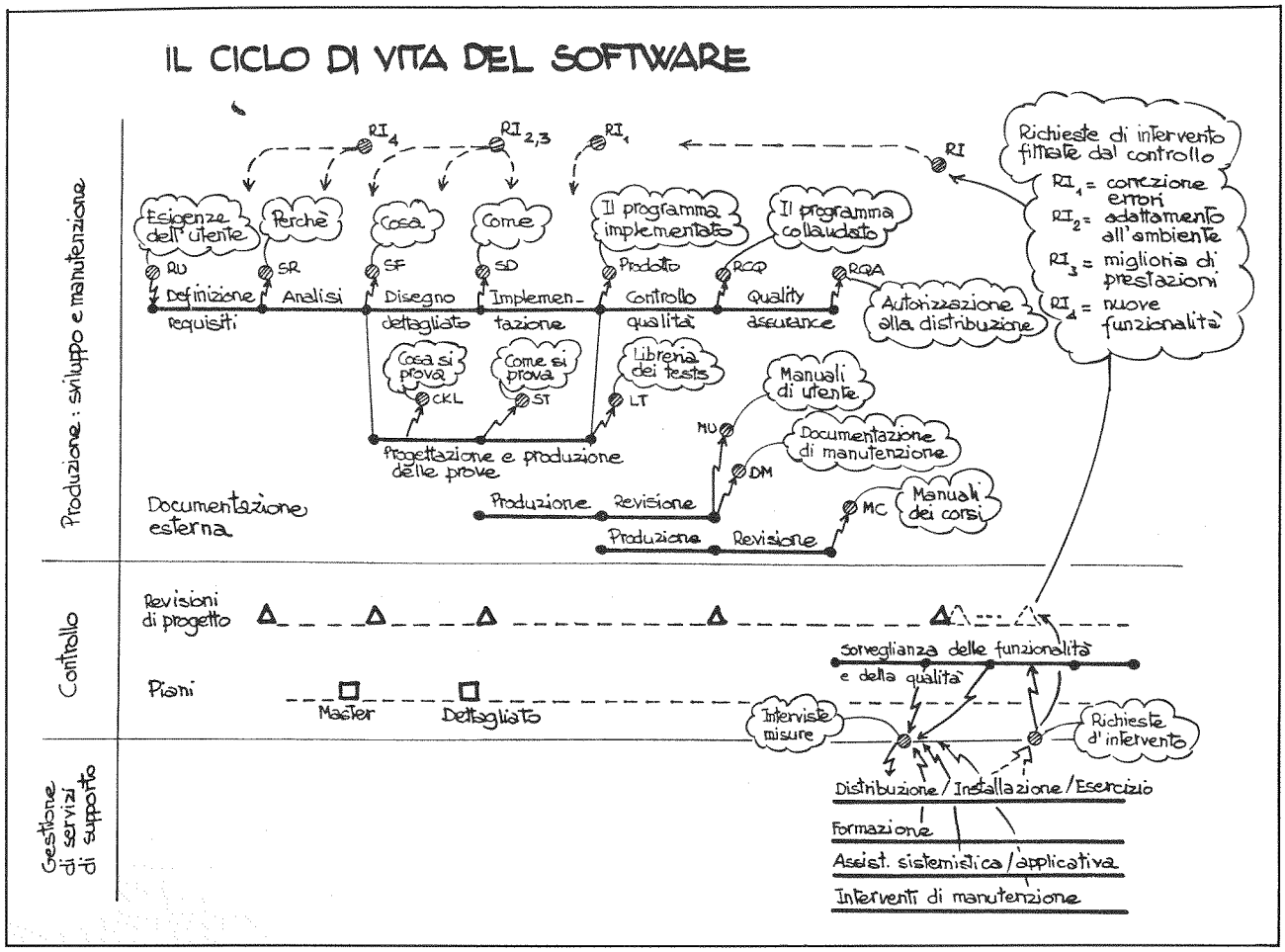


Fig. 6

4. GLI STANDARDS DI DOCUMENTAZIONE TECNICA: CONTENUTI

4.1 Premessa: enfasi sugli obiettivi e le motivazioni

Lo schema di contenuto dei vari documenti che saranno presentati è quello del documento (METO 31).

Si insisterà soprattutto sui *motivi* che spingono a specificare un dato argomento in un documento piuttosto che in un altro, con un costante richiamo degli obiettivi di *qualità* (vedi ancora Fig. 2) e di *produttività* che sono alla base degli standards proposti in (METO 31) (Fig. 7).

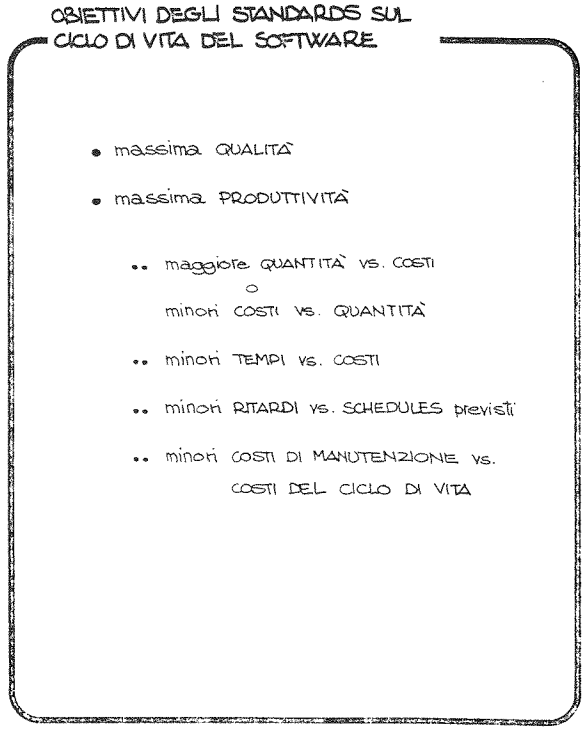


Fig. 7

Questo duplice obiettivo di *qualità/produttività* è perseguito in questi standards con il criterio generale delineato in Fig. 8.

Ogni documento deve servire

1) a specificare le informazioni dettagliate di input per la *fase di progetto immediatamente successiva*

2) ad anticipare, anche solamente *in linea di massima*, i *problemi, vincoli* e *soluzioni* che sono attualmente noti su *tutto il ciclo di vita*, in modo da delimitare la strada intrapresa nei valori di dettaglio ed evitare di pensare a soluzioni incompatibili (per motivi tecnici o finanziari o di schedule) con i vincoli noti.

3) a supportare una attività di *verifica e validazione* delle soluzioni adottate in funzione del livello di qualità voluta e dei vincoli di costi, prima di procedere in ogni fase successiva.

4.2 Limiti della trattazione

In questo lavoro si espone il contenuto della documentazione tecnica del Software e si faranno riferimenti al suo utilizzo nelle attività di controllo e gestione servizi; non sarà invece trattato il contenuto dei documenti di pianificazione, anche se spesso sarà fatto ad essi riferimento.

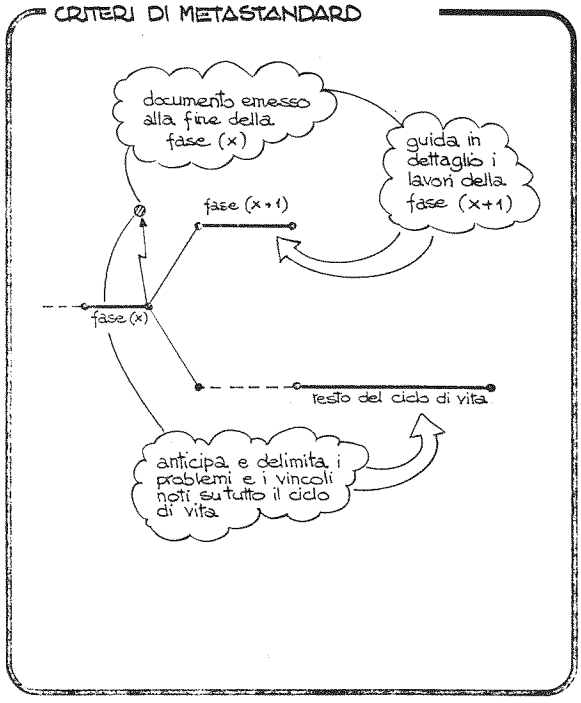


Fig. 8

4.3 Le richieste dell'Utente (RU)

4.3.1 Contenuto

Le richieste dell'utente (RU) sono il documento che dà il via a tutto il progetto.

Si tratta nella più parte dei casi di un documento informale, non standardizzato (una lettera, una nota) in cui l'utente (d'ora innanzi per utente si intenderà sia l'utente che comissiona un prodotto sia il management di una azienda produttrice di sistemi o applicazioni o il relativo marketing) esprime con il suo linguaggio le esigenze di nuove applicazioni o di migliorie alle applicazioni già in uso.

L'obiettivo – economico o organizzativo – è qui sancito in modo breve e sintetico: una analisi completa e professionale di esso è demandata al documento successivo (SR).

Il contenuto indirizza la attività di definizione dei requisiti e può essere tenuto presente nella stesura dei manuali e dei corsi.

4.3.2 Responsabilità

Chi lo scrive: l'utente

Chi lo usa: l'utente e l'analista che lo aiuta nel formalizzare le SR.

4.4 Le Specifiche di Richieste (SR)

Nella storia di un prodotto Software si può riscontrare che la descrizione accurata delle sue funzionalità (il "cosa" fa un certo prodotto – vedi dopo le SF) è, prima o poi, sempre disponibile.
Anche nei casi in cui i progettisti abbiano trascurato l'emissione di un documento accurato e completo di SF, interviene il Manuale di utente a supplire a questa carenza.

È invece assai raro riscontrare l'esistenza di un documento che chiarisca *i motivi* che hanno dato origine a un dato prodotto.

Alcuni di questi motivi sono evidenti nelle funzionalità stesse e sono spiegati nel manuale: spesso tuttavia una buona parte degli obiettivi organizzativi, economici e tecnologici di una applicazione EDP o di un nuovo prodotto di Software di base rimangono nella mente dei managers e degli analisti che li hanno concepiti.

Invece la formalizzazione di detti *obiettivi* è vitale per

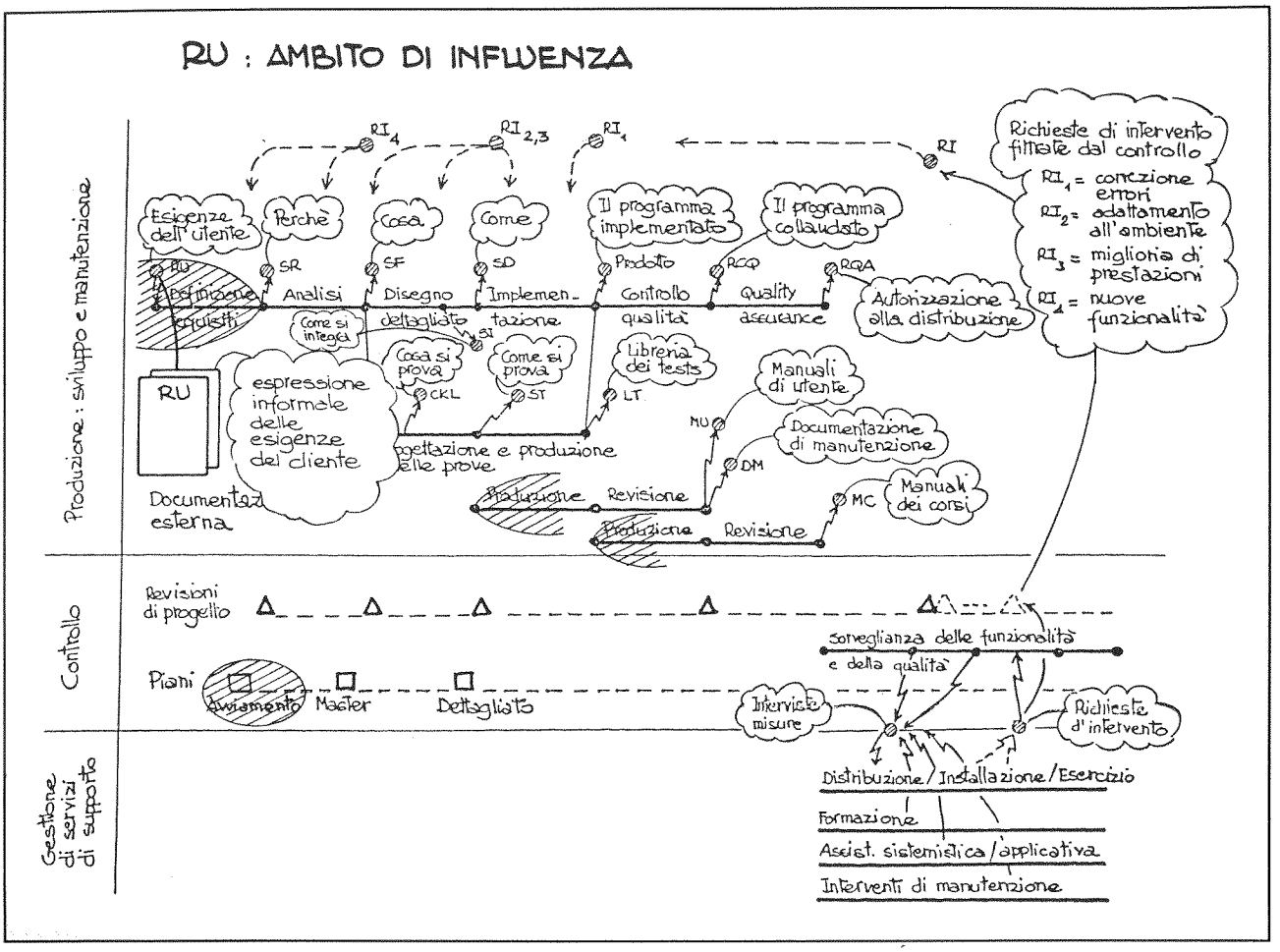


Fig. 9

tutta la vita del prodotto. Si sa che in media i costi di modifica e di estensione di un prodotto oscillano tra il 50% e il 75% del costo dell'intero ciclo di vita: ma ciò è spesso causato dalla volontà di aggiungere ai programmi esistenti funzionalità non coerenti con essi, per le quali questi programmi non erano proprio stati pensati.

In questi casi una lettura di un documento sugli obiettivi farebbe capire che certi limiti non erano una dimenticanza, ma che erano stati scientemente accettati per ridurre i costi, e che un rifacimento invece che un'estensione potrebbe essere la via corretta.

A questo scopo vogliono soddisfare le SR, che devono descrivere in modo completo e organizzato tutto ciò che solo in forma germinale era contenuto nelle RU.

4.4.1. SR: Contenuti

Presentiamo qui un possibile indice standard di documento di specifiche di requisiti. Lo standard precisa i contenuti del documento, non gli strumenti linguistici da utilizzare: come si può vedere nel documento "S. Cappelli – L'uso di reti di Petri per specifiche funzionali: un esempio reale – obiettivo METOD", molto del contenuto può essere specificato con l'ausilio di reti di Petri.

Si osservi che, con lo scopo di rendere della medesima generalità l'uso di un tale standard, si sono inclusi aspetti (come ad esempio di esigenze di mercato) che spesso sono assenti (o per lo meno non sono presenti in un modo evidente) in moltissimi ambienti di sviluppo software.

Passeremo in rassegna nei prossimi moduli ciascuno dei capitoli indicati, illustrandone obiettivi e motivazioni e dettagliandoli, ove necessario, in ulteriori paragrafi (fig. 10).

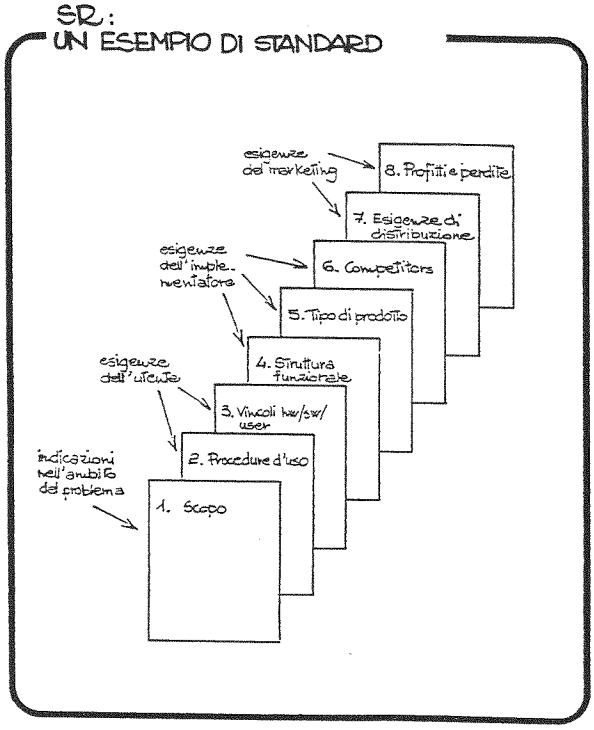


Fig. 10

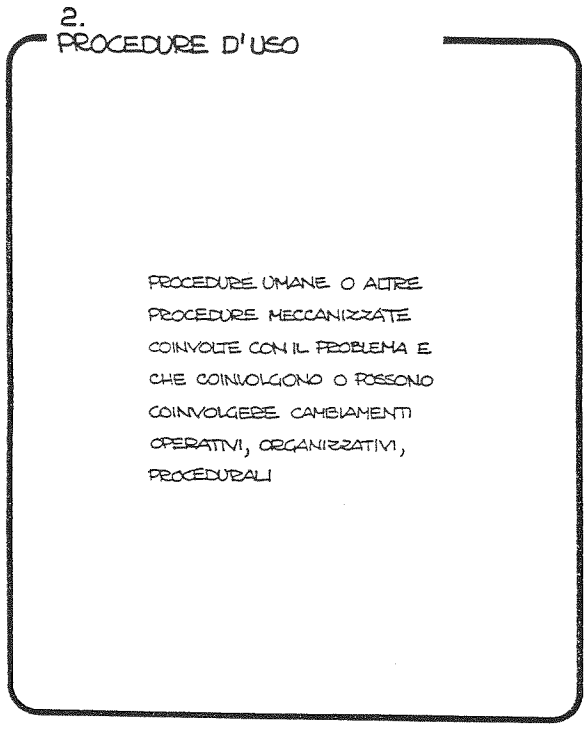


Fig. 10-2

1. Scopo

Il primo capitolo deve fornire la chiave di lettura dell'intero documento: a tale scopo esso deve fornire, in modo chiaro ma sintetico, un'immagine della situazione attuale dell'ambiente in cui si vuole intervenire, una esplicitazione dei problemi, delle difficoltà o delle esigenze attualmente sentite ed una ipotesi di stato auspicato nel sistema modificato. (Fig. 10.1)

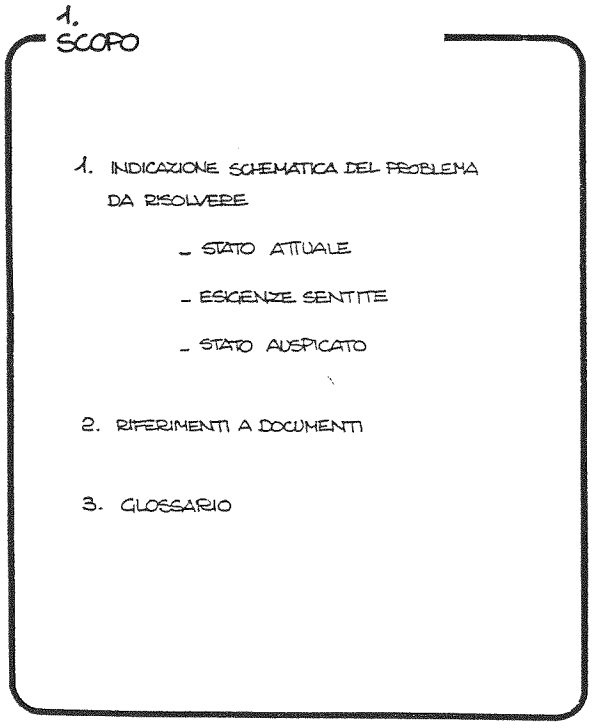


Fig. 10-1

Il riferimento a documenti tecnici interni e non, a testi, note, letteratura utile per chiarimenti sul problema e sulla soluzione, e un glossario che elenchi tutti i termini "gergali" o comunque non di universale significato completano questo capitolo.

2. Procedure d'uso

La descrizione delle procedure d'uso è sostanzialmente la descrizione del comportamento attuale del sistema, eventualmente descritto mediante reti di Petri, e del comportamento desiderato del sistema modificato (fig. 10.2).

Questa è una delle parti centrali del documento.

Nel descrivere il comportamento "desiderato" del sistema modificato, mettere in luce i motivi che originano le richieste di modifiche.

L'aver evidenziato bene queste "motivazioni" può essere di ausilio decisivo nel valutare gli impatti organizzativi e tecnici di future ulteriori modifiche che i manutentori o estensori del progetto dovessero fare in seguito.

3. Vincoli HW/SW/USER.

Questo capitolo ha il compito di dettagliare tutti i vincoli derivanti dall'ambiente in cui il prodotto dovrà funzionare: tali vincoli possono dipendere sia dallo hardware impiegato (periferiche, memoria, ecc.), sia dal software (sistema operativo, integrazione con sottosistemi o con altri prodotti, ecc.), ma soprattutto dall'ambiente utente, che può richiedere specifiche caratteristiche di operabilità e di costi di esercizio e di tempi di disponibilità, dal punto di vista dell'impiego del prodotto, e che può richiedere precise caratteristiche per la manutenzione e la continuazione dello sviluppo. (fig. 10.3)

4. Struttura funzionale

Un documento di requisiti dovrebbe limitarsi ad esaminare il comportamento desiderato, ovvero ad individuare con dettaglio il problema, senza fornire soluzioni, che dobbono invece essere dettagliate in documenti di specifiche funzionali. (fig. 10.4)

Tuttavia, come vedreamo anche in altri documenti, è opportuno che una "corsa in avanti" sia presente:

> inseriamo quindi nello standard un cenno a quale potrebbe essere una proposta di funzionalità da richiedere al prodotto: ciò potrà garantire l'estensore delle specifiche funzionali che il problema non è "campato in aria", ma ha già avuto una verifica di fattibilità; potrà essere un valido suggerimento di avvio per lo studio successivo; ma potrà anche essere un insieme di informazioni che l'analista, a ragion veduta, decide di modificare completamente.

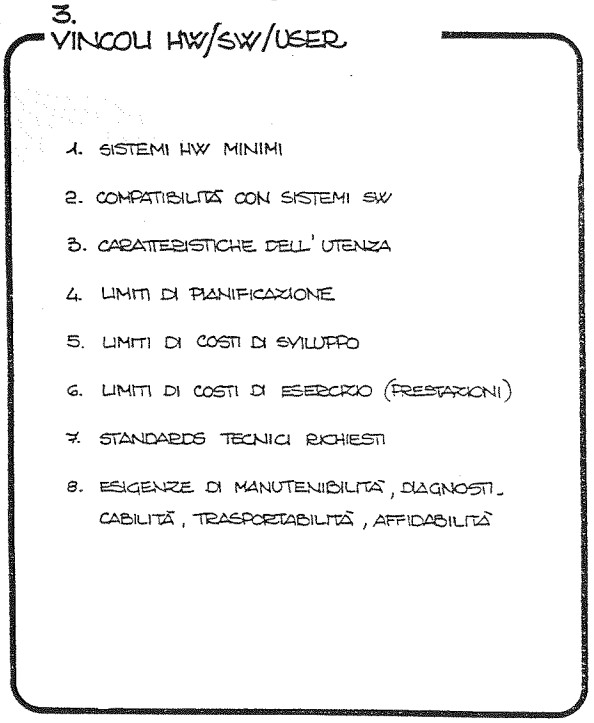


Fig. 10-3

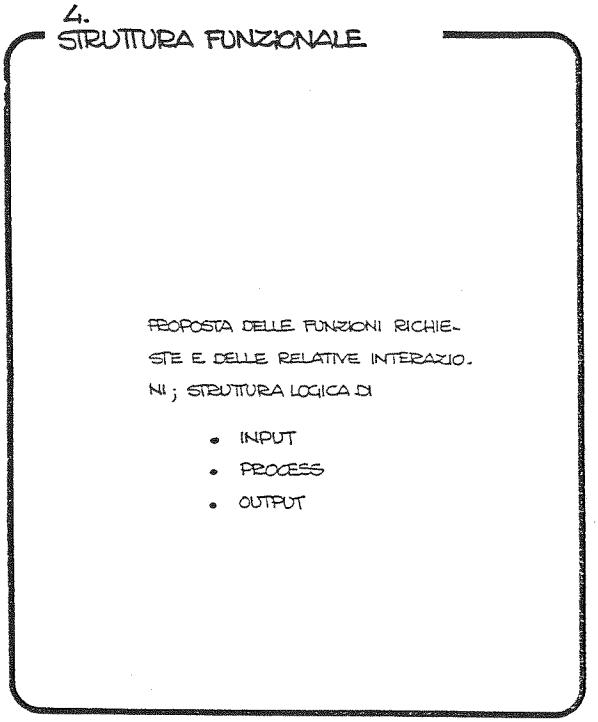


Fig. 10-4

5. Tipo di prodotto

Caratteristiche diverse di prodotto impongono problematiche diverse di sviluppo, di gestione, di distribuzione. (fig. 10.5)

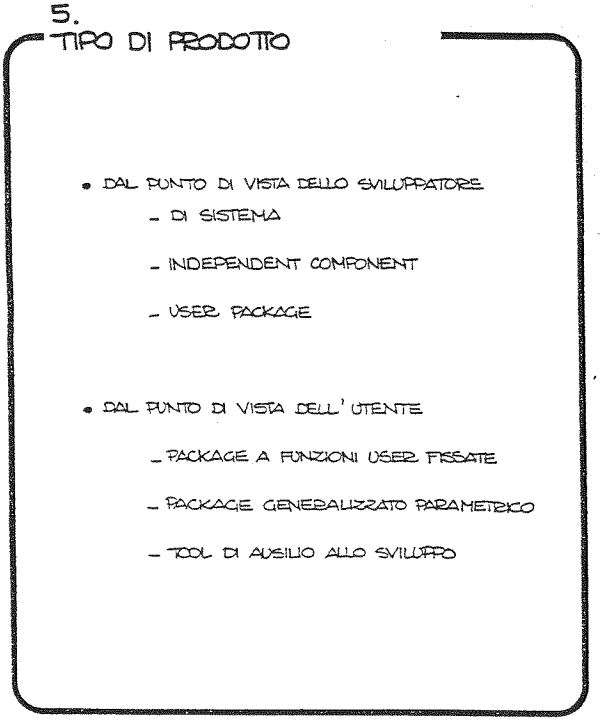


Fig. 10-5

È per questo che è molto importante chiarire bene la collocazione del prodotto, sia dal punto di vista dell'ambiente di sviluppo, sia da quello dell'utente.

Un prodotto può essere:

una parte di sistema (si pensi ad una porzione indispensabile di un nucleo di un sistema operativo, o anche ad una porzione indispensabile di gestione di una data base in un sistema edp): in tal caso le problematiche di integrazione, di coerenza con gli standard e di controllo qualità integrato verranno enfatizzate;

un componente indipendente (si pensi ad un compilatore, nel caso di un sistema operativo, o ad una procedura di fatturazione nel caso di un sistema edp): in questo caso gli aspetti di pianificazione potranno avere una integrazione meno enfatizzata, non influenzando lo sviluppo di altri progetti;

un package (si pensi ad una procedura di contabilità fornita da chi sviluppa un sistema operativo ai suoi clienti): in questo caso è da evidenziare l'aspetto di standard nella documentazione, la operabilità, ecc.

Anche dal punto di vista dell'utente una classificazione come quella riportata può richiedere enfasi su aspetti produttivi differenti da caso a caso.

6. Competitors

Difficilmente si scoprono problemi nuovi: le esigenze sentite in un ambiente sono spesso connaturate con le caratteristiche di quell'ambiente, che sono comuni a molti altri ambienti.

Quindi, difficilmente si inventato soluzioni nuove.

Allo scopo di evitare lavoro aggiuntivo, soluzioni di livello e qualità inferiore a quelle già esistenti, soluzioni contradditorie con altre, prestazioni inferiori, ecc. (oltrechè, qualora il prodotto da sviluppare sia da rivedere, un prodotto decisamente inferiore a quelli esistenti sul mercato) è opportuno studiare ciò che già sia stato fatto nel campo. (fig. 10.6)

7. Esigenze di distribuzione

Il tipo di prodotto e la sua destinazione possono determinarne la fisionomia architetturale e funzionale: è pertanto molto importante specificarne la destinazione (fig. 10.7):

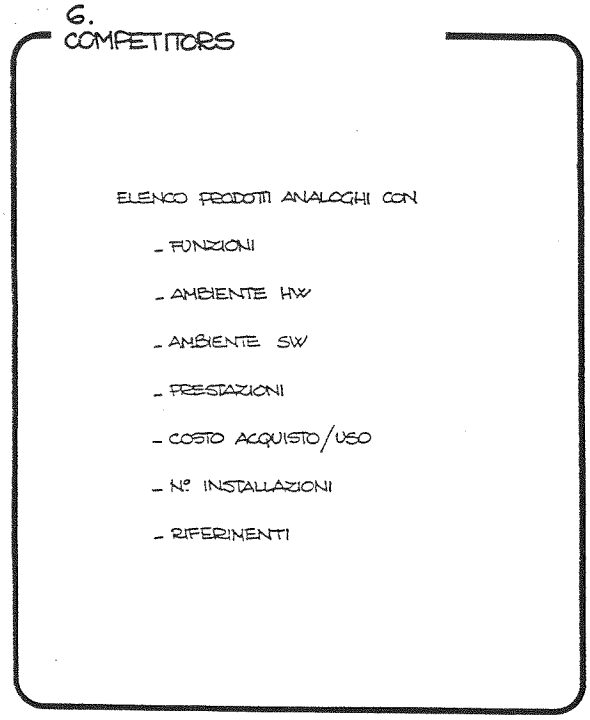


Fig. 10-6

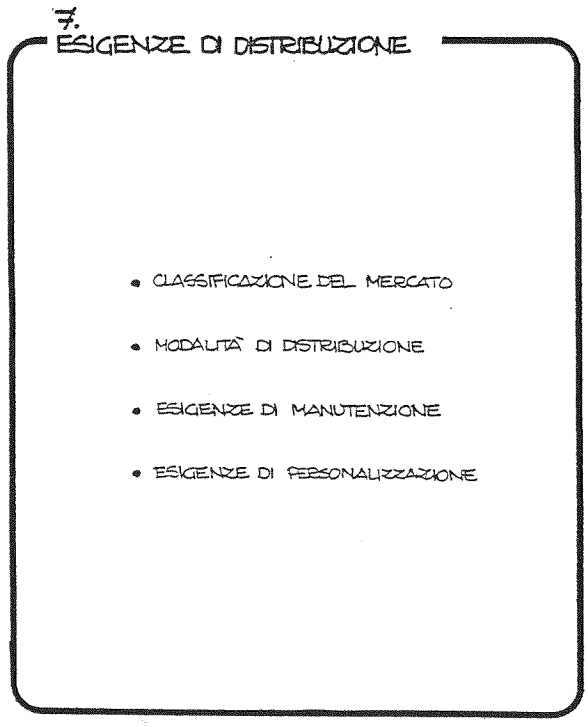


Fig. 10-7

- un prodotto da distribuire richiede una organizzazione di software factory atta alla manutenzione ed alla generazione del prodotto per ogni nuovo utente;

- un prodotto che venga ceduto "chiavi in mano" deve avere una accurata capacità di autocontrollarsi in modo estremamente operabile;

- un prodotto che richieda personalizzazioni presuppone fasi di installazione che pongono problematiche ancora differenti.

8. Profitti e perdite

Raramente un prodotto viene costruito perchè è inevitabile o per prestigio: la motivazione è sempre di tipo economico.

Raramente la valutazione economica è esplicitata e chi chiede un prodotto ha una chiara visione dei vantaggi economici che gli derivano.

Tuttavia è ragionevole proporre nel documento un capitolo che faccia riferimento a tali temi, da cui, nel caso di rivendita di prodotti, è completamente individuabile una fascia di mercato ed una strategia di vendita.
Si osservi che la mancanza di tali indicazioni non è determinata dall'impossibilità di darle, ma piuttosto dalla diseducazione diffusa, che tende ad ometterle; in particolare le prime due voci sono di misura o di stima non difficile. (fig. 10.8)

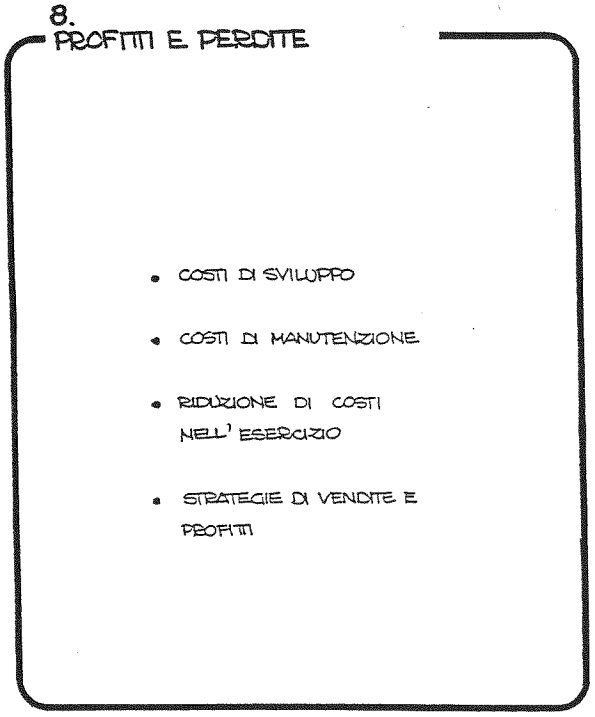


Fig. 10.8

SR: Ambito di influenza

Le Specifiche dei Requisiti hanno una influenza diretta sulla fase di analisi che ne dovrà tenere conto come elemento guida per la definizione delle funzionalità.

Possono contribuire alla *definizione dei tests,* soprattutto quelli di accettazione se essi sono impostati con la ottica di verificare che i requisiti applicativi sono suportati dal prodotto; possono contribuire all'impostazione dei *manuali* di *utente* e dei *corsi,* guidandone la presentazione delle motivazioni del prodotto e la scelta degli esempi; sono il riferimento nelle attività di *sorveglianza della qualità*; e sono l'oggetto della 1ª revisione di progetto. (fig. 11, alla pagina seguente).

SR: responsabilità

Chi le scrive: gli analisti di sistema e/o di organizzazione, in stretta collaborazione con l'utente/committente; o il committente stesso;

Chi le usa : analisti di sistema, collaudatori, technical writers.

4.5 Le specifiche funzionali (SF)

Esamineremo qui un esempio di standard di documentazione di specifiche funzionali, discutendo l'indice proposto ed i contenuti di ciascun paragrafo. (fig. 12)

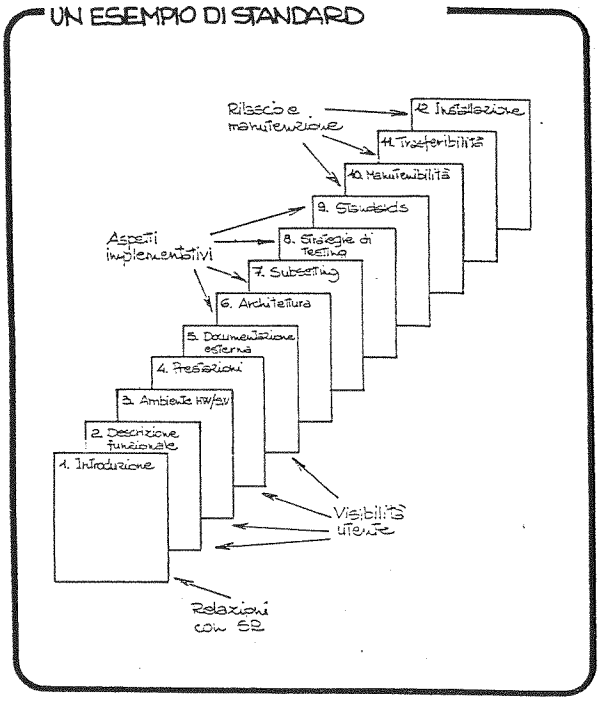


Fig. 12

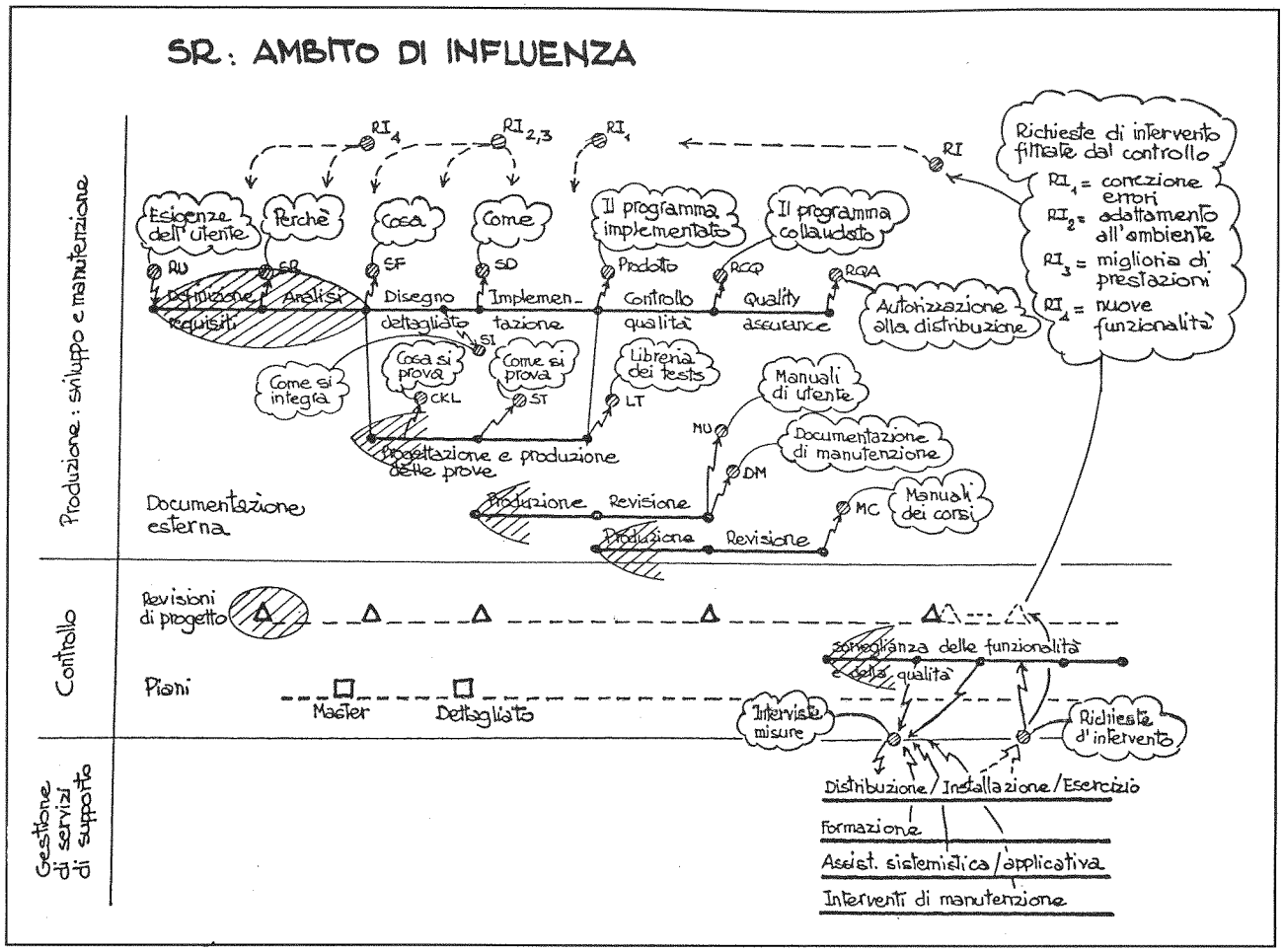


Fig. 11

Osserviamo innanzitutto che tale documento deve contenere:

un preciso riferimento al precedente documento di specifiche di requisiti;

una specificazione completa sul modo in cui l'utente interagirà con il prodotto (le "funzionalità") con particolare enfasi sulla facilità di uso;

la definizione dell'architettura del prodotto (allo scopo di poter costruire un piano "master" dello sviluppo, e di verificare in modo definitivo la fattibilità delle funzionalità volute);

la specificazione completa degli obiettivi di prestazioni e dei requisiti e modalità di prova, di rilascio e distribuzione, di manutenzione, ai fini di garantire e supportare al meglio questi aspetti di qualità con un disegno adeguato.

4.5.1 SF: Contenuti

1. Introduzione

L'introduzione ha lo scopo di ricordare al lettore quale è il problema che il prodotto specificato è chiamato a risolvere: il capitolo è da considerare soprattutto un riferimento alle specifiche di requisiti e a documenti rilevanti per la comprensione della soluzione proposta (fig. 12.1).

2. Descrizione funzionale

La descrizione funzionale è una delle parti più rilevanti del documento; essa può essere articolata in diversi paragrafi (fig. 12.2.):

la descrizione delle funzioni, ovvero delle procedure di interazione dell'utente con il prodotto: tipicamente può trattarsi della Rete di Petri che descrive il comportamento della procedura (non dell'ambiente organizzativo);

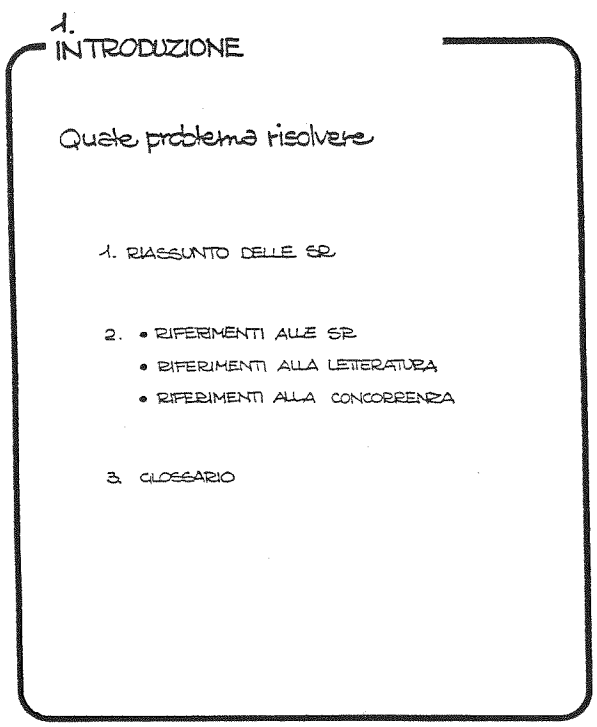


Fig. 12-1

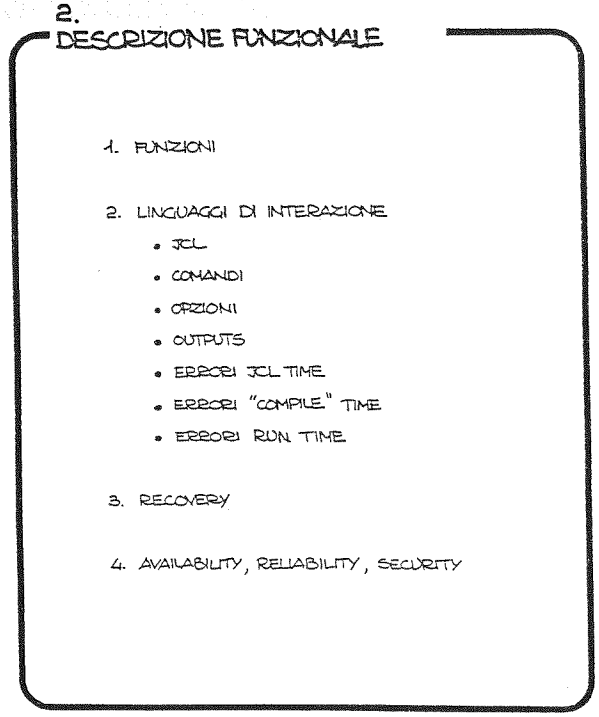


Fig. 12-2

- la descrizione dettagliata (compreso una sintassi formale, ove possibile) di tutte le modalità di interazione: formato di dati di ingresso, istruzioni di lancio e formato di maschere, ecc.;

- la specificazione dettagliata delle modalità operative per la recovery, ovvero per il ripristino di situazioni operabili a fronte di condizioni di errore;

- la specificazione dettagliata di aspetti di availability (ovvero, disponibilità del sistema, eventualmente a fronte di uso errato o a degradazione delle funzioni a causa di guasti), di reliability (ovvero affidabilità di sistema, spesso, nel SW, legata ad un efficiente controllo di qualità) di security (ovvero, di sicurezza di sistema, a fronte di usi che non si vogliono consentire), di integrity (ovvero, di integrità delle informazioni, protette da distruzioni involontarie o dolose).

3. Ambiente HW/SW

Le specifiche funzionali devono descrivere anche le condizioni ambientali di HW e SW necessario (fig. 12.3):

- sia per indicare le risorse minime necessarie per supportare il sistema;

- sia per indicare le risorse massime supportabili dal sistema.

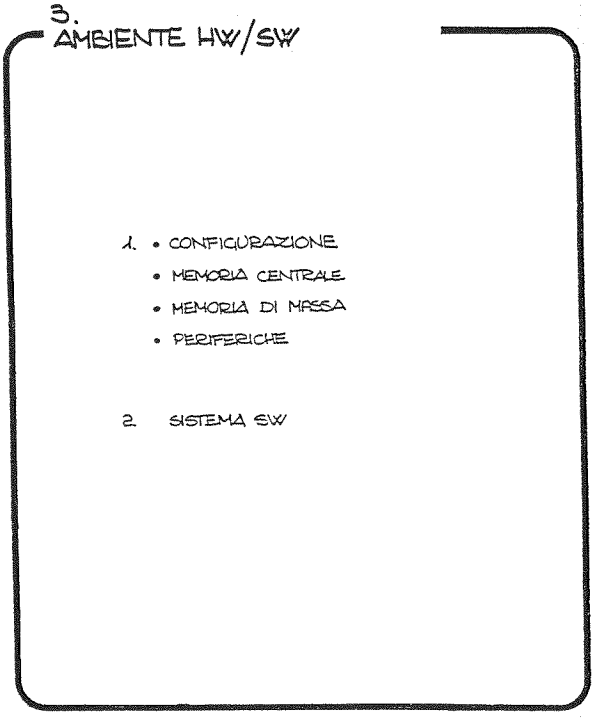


Fig. 12-3

4. Prestazioni

Anche gli obiettivi di prestazioni sono un attributo su cui le specifiche funzionali devono fornire indicazione: è importante sottolineare come sia indispensabile fornire, a fianco delle indicazioni di occupazione e di tempi di esecuzione, dei precisi criteri per la misura di tali prestazioni (eventualmente con l'indicazione de "tools" necessari) senza i quali le affermaizioni perdono un qualunque valore. (fig. 12.4)

5. Documentazione esterna

La documentazione esterna deve essere indicata, non solo per qualificare completamente le caratteristiche e l'ambiente di utenza del prodotto, ma anche per permettere una completa pianificazione delle attività. (fig. 12.5)

6. Architettura

L'architettura del sistema, senza arrivare al massimo dettaglio, permette di (fig. 12.6):

. verificare la fattibilità;

. distribuire le attività di sviluppo;

. costruire in piano e stimare tempi e costi di sviluppo.

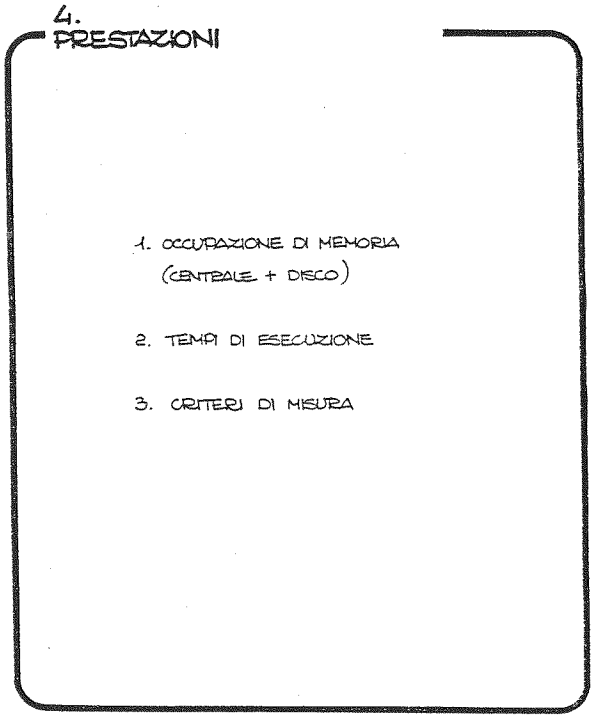


Fig. 12-4

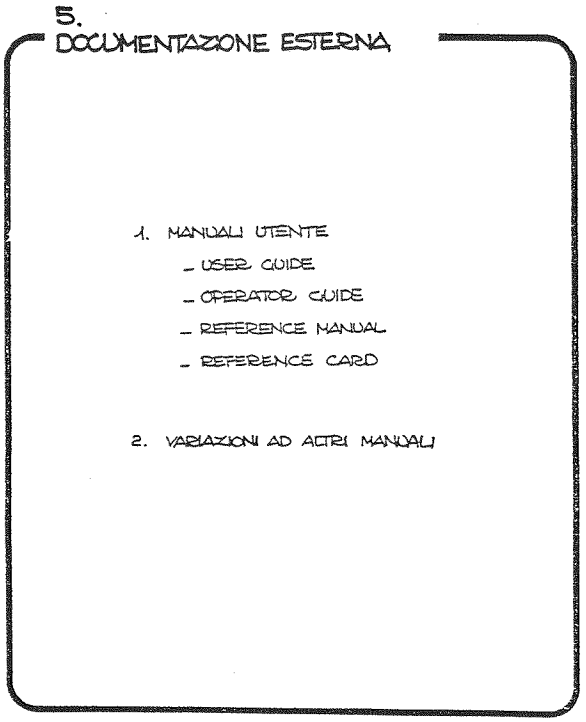


Fig. 12-5

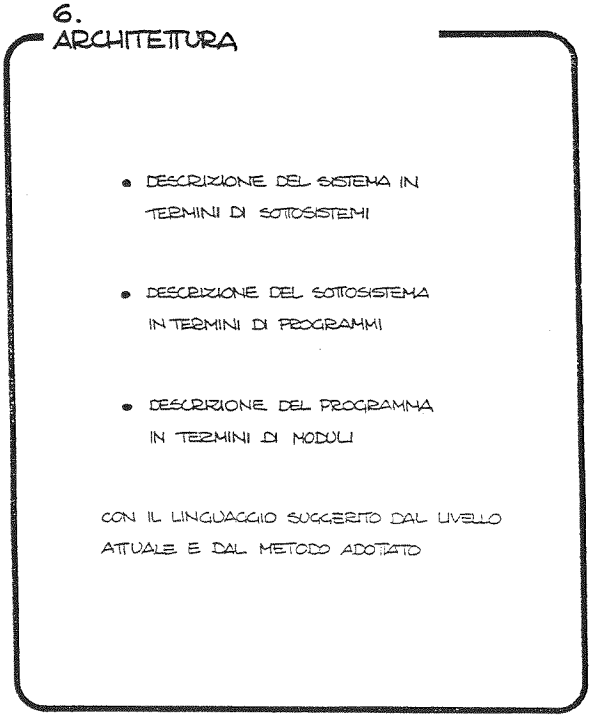


Fig. 12-6

Poichè è atteggiamento generale della progettazione strutturata top-down ricavare la struttura architetturale di un sistema dalla struttura funzionale (ciò garantisce, in genere, costi di modifica "proporzionali" alla "distanza" funzionale delle modifiche richieste dalla situazione presente), questo capitolo risente solitamente in modo molto massiccio del paragrafo 2.1.

Le informazioni da specificare in questo capitolo si possono così riassumere:

- struttura gerarchica del prodotto in componenti, descritti in modo statico nelle loro funzionalità, ai vari livelli di gerarchia
- i dettagli sui dati di intefaccia dei componenti tra di loro e con il sistema ospite.
- la descrizione della "dinamica" di funzionamento.

7. Subsetting

Le specifiche funzionali hanno l'obiettivo di descrivere il prodotto nella sua versione finale definitiva: molto spesso tale versione è pianificata molto lontano nel tempo, e il costruttore (o l'utente) ha esigenza di utilizzare versioni ridotte con molto anticipo (anche in corrispondenza di strategie di mercato). (fig. 12.7)

Una suddivisione delle funzionalità in tal senso è quindi fondamentale per una corretta pianificazione.

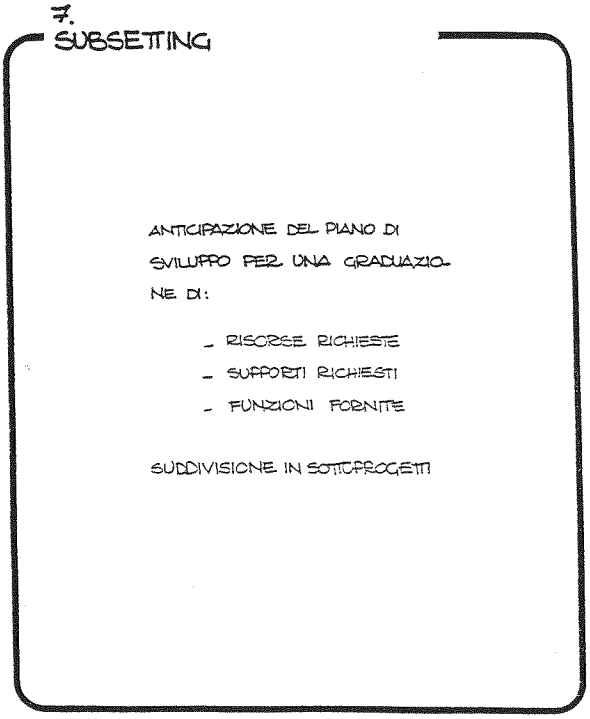


Fig. 12-7

È necessario inoltre comprendere bene i motivi di questa necessità di "completezza" delle specifiche funzionali, anche se detta completezza sarà realizzata in tempi lunghi: la specificazione accurata degli obiettivi funzionali finali consente di progettare una crescita graduale delle funzionalità realizzando versioni intermedie del prodotto, in modo che il passaggio da una versione alla successiva avvenga attraverso estensioni (anche esse "progettate") e non attraverso rifacimenti completi (spesso provocati dal fatto che il progetto delle versioni iniziali non ha tenuto conto dell'obiettivo finale).

Subsetting: esempio

L'esempio illustrato mostra, in modo astratto, come un prodotto possa essere qualificato e distribuito in una prima versione S1, le cui ultime fasi di produzione sono sovrapposte allo sviluppo della nuova estesa versione S2. (fig. 12.7.1)

8. Strategie di testing

La definizione delle modalità di effettuazione del controllo qualità è in grado di influenzare lo sviluppo sia per le esigenze di strumenti che permettano controlli automatici (esigenze di comparatori di files, flaggers, ecc.), sia per modifiche architetturali con lo stesso scopo (simulazione di terminali via disco, ecc.). (fig. 12.8)

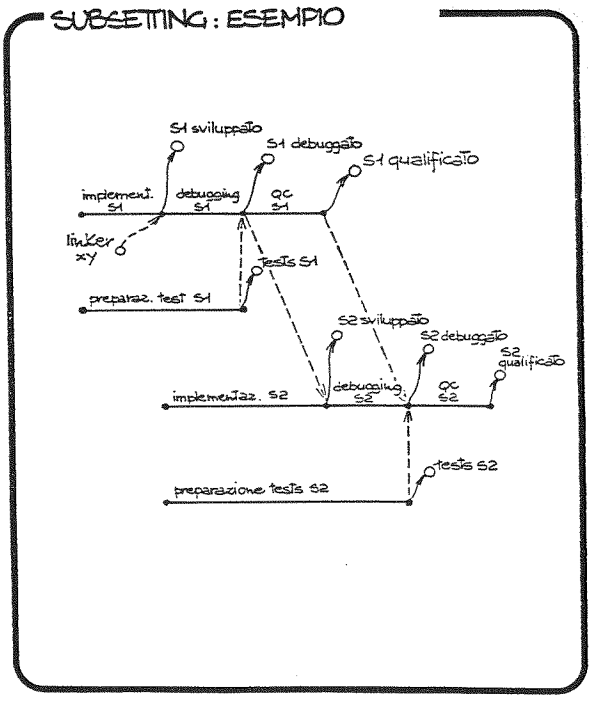


Fig. 12-7-1

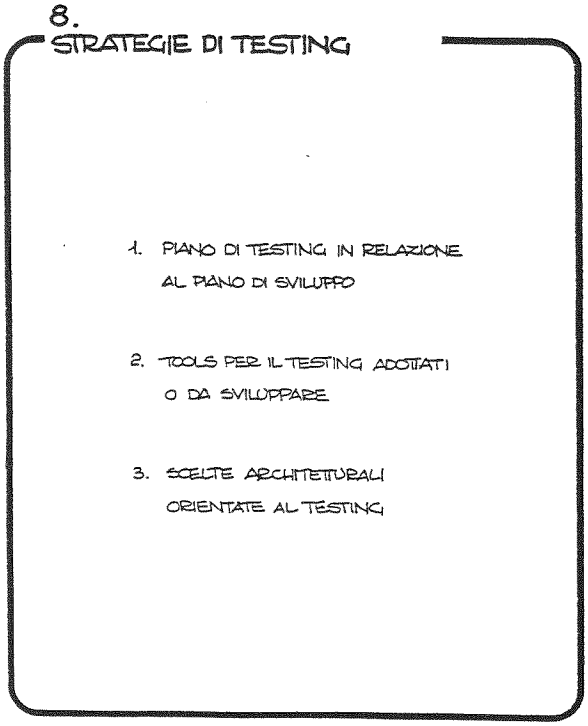


Fig. 12-8

Non è possibile una valutazione dei tempi e dei costi di sviluppo senza una chiara visione di tale strategia.

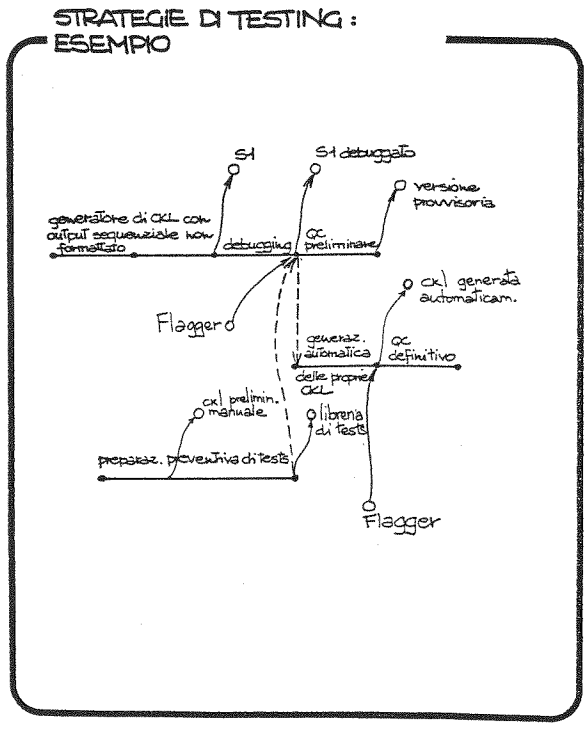


Fig. 12-8-1

Strategie di testing; esempio

L'esempio mostra un caso particolare di sviluppo di un tool per il testing, (generatore di checklists) in cui la strategia di testing usa il prodotto stesso, sovrapponendo un controllo qualità preliminare (per anticipazione di problemi) al controllo di qualità che usa il prodotto stesso. (fig. 12.8.1)

9. Standards

Tutti gli standards operativi, gestionali, di documentazione devono essere elencati (fig. 12.9)

Ad esempio, lo standard delle specifiche funzionali, lo standard di accesso ai calcolatori o di uso delle librerie di supporto, lo schema di ciclo di sviluppo del SW adottato, le modalità di consegna del prodotto finito.

Lo scopo di questo paragrafo non è tanto una elencazione burocratica di regole, quanto un autostimolo per il gruppo di progetto a riflettere sui metodi da adottare e a prevederne in modo esplicito l'uso, scegliendo e indicando i metodi e standards più adatti alle varie fasi.
Ciò è affermato nell'assunzione che una mancata indicazione degli standards adottati è la migliore premessa al mancato uso di un metodo (quale che sia), con la conseguente gestione disordinata e costosa del progetto stesso.

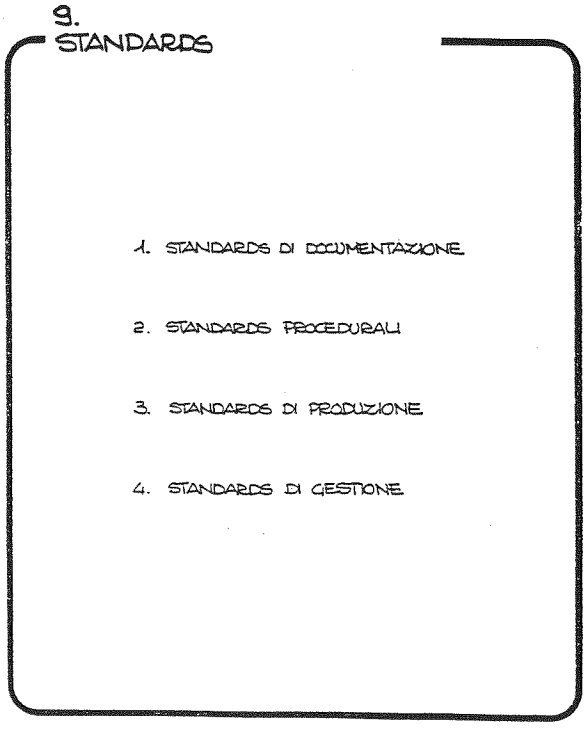


Fig. 12-9

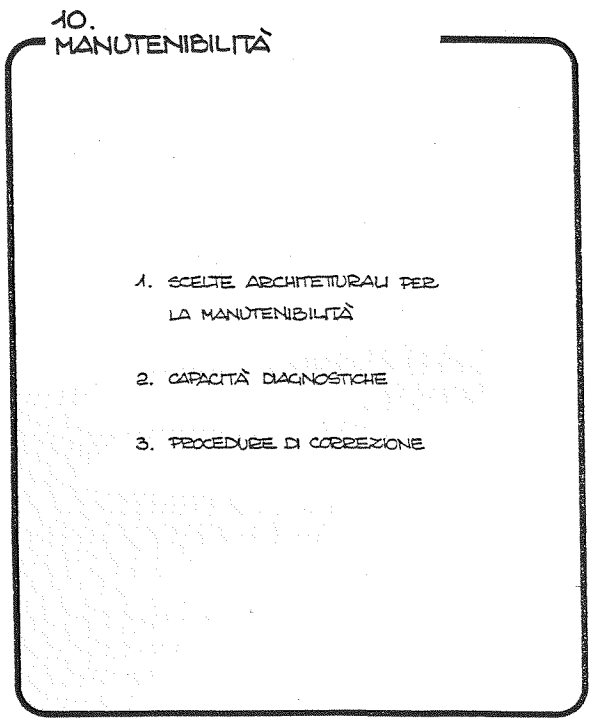


Fig. 12.10

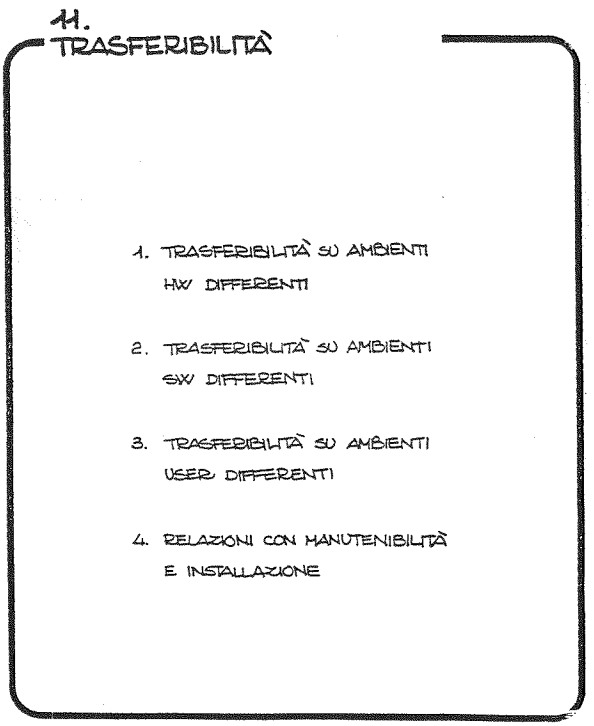


Fig. 12.11

10. Manutenibilità

La manutenibilità di un prodotto non è solo legata a scelte architetturali, (coesione funzionale di moduli, information hiding, ecc.), ma anche a scelte implementative di dettaglio (come la presenza di istruzioni sotto compilazioni condizionali per un "debudding permanente", la presenza di moduli "dummy" per eventuali sviluppi futuri, ecc.) e gli aspetti operativi specifici per effettuare le correzioni o le distribuzioni di nuove funzioni (via patches fornite su dischetto o disco, via comunicazioni telefoniche, via rete, ecc.). (fig. 12.10)

11. Trasferibilità

Gli aspetti di trasferibilità sono rilevanti in diversi ambiti:

- è necessario dichiarare quale sarà il grado di portabilità su macchine con differente struttura o configurazine HW: ovvero, quanto il programma risente della struttura delle macchine (moltissimo se il programma è in assembler, meno se in linguaggio ad alto livello) e quanto del tipo e delle caratteristiche delle periferiche, della memoria, ecc.;

- è necessario dichiarare il tipo di software di supporto necessario (linguaggio in cui il programma è scritto, caratteristiche del sistema operativo, ecc.), orientando la descrizione all'eventualità di uso del prodotto in altri ambienti;

- è necessario verificare e dichiarare l'eventuale adeguatezza del prodotto a differenti tipi di utenza o problemi degli utenti (funzioni che risentono di aspetti legislativi nazionali, linguaggi di interazione con l'operatore più o meno sofisticati, ecc.).

Gli aspetti succiatati sono in evidente relazione con aspetti, architetturali, orientati alla manutenibilità e all'installazione. (fig. 12.11)

Trasferibilità: Esempi

La trasferibilità può avere impatti sull'architettura in diversi modi; tra l'altro:

è pensabile il ricorso a tecniche di information hiding, che incapsulino completamente tutti gli aspetti "fisici" di alcune gestioni (ad esempio il trattamento delle periferiche) mostrando al programma che le usa solo aspetti sintattici; ciò sia in relazione a problemi di configurazione HW, sia a problemi di uso di primitive SW (accessi a files di organizzazioni non standard, ecc.);

è pensabile a organizzazioni del prodotto che permettano tecniche di boostrap, che riducano il costo del trasferimento (sia nuclei di prodotto che servono a sviluppare estensioni del prodotto stesso, come ad esempio il nucleo di un interpreter LISP; sia trasferimenti di prodotti, come largamente mostrato dalla letteratura Pascal);

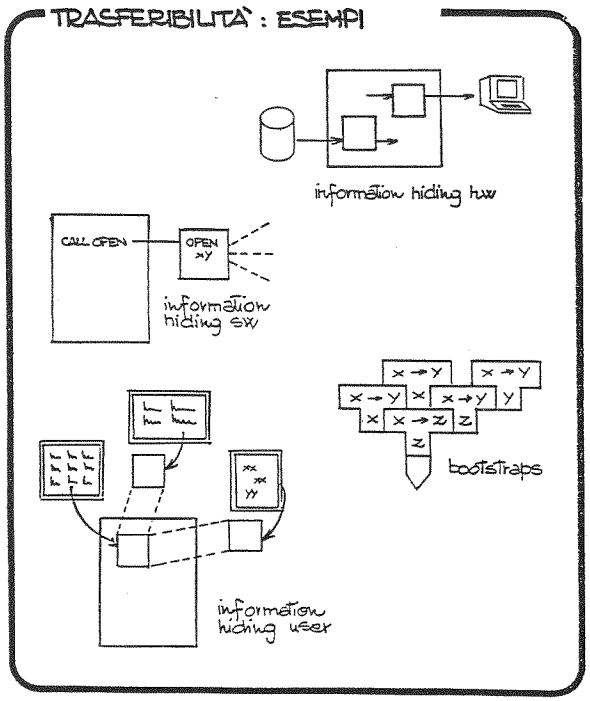


Fig. 13

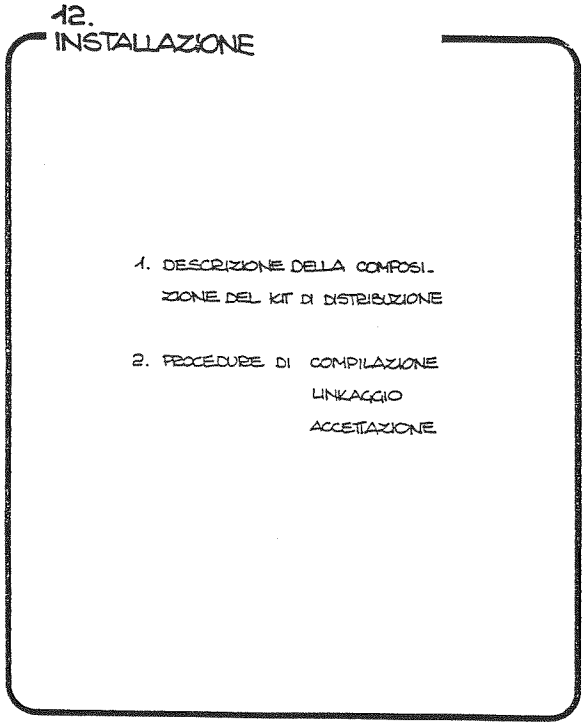


Fig. 14

. è pensabile ad una parametrizzazione del prodotto via tabelle o altro (ad esempio, il problema delle "lingue nazionali" nel linguaggio di interazione) (fig. 13).

12. Installazione

L'installazione è una fase estrememante delicata dal punto di vista del mercato (fig. 14):

. deve essere facilmente disponibile l'insieme di tutti gli strumenti e di tutti i documenti per installare il prodotto su una macchina;

. deve essere preso in esame e dichiarato il modo di distribuzione (via sistemisti, via posta, via telefono, via rete, ecc.);

. il kit di distribuzione deve essere evidentemente affiancato dalla precisa indicazione delle procedure operative di personalizzazione, costruzione, accettazione.

Installazione: esempi

Ad esempio, l'installazione può prevedere la presenza di programmi sorgente da compilare (o comunque preelaborare) mediante programmi forniti nella sola forma di oggetto, per ottenere il prodotto finale. (fig. 14.1)

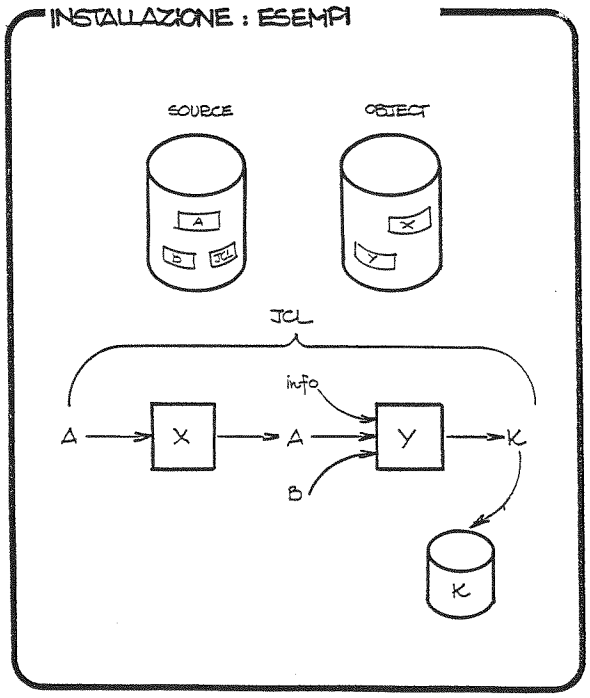


Fig. 14.1

SR: ambito di influenza

Le specifiche funzionali sono il documento chiave di tutto il progetto, determinando e guidando tutte le attività successive. (fig. 15)

Esse forniscono le informazioni di base per il *disegno di dettaglio* del prodotto nonchè per la stesura delle checklists funzionali e la *progettazione dei tests*. Sono il documento di partenza per la produzione dei *manuali* e dei corsi. I collaudatori e i manutentori le usano come riferimento per individuare eventuali incongruenze nei manuali o nella documentazione di disegno.
Specialmente le informazioni sull'architettura, le misure di prestazioni e la provabilità consentono di stendere un accurato *piano MASTER*, il quale fornisce costi e date di massima per tutto l'arco del progetto, e programma invece in dettaglio la fase di disegno di dettaglio e di stesura della checklist e specifiche di test.
SF e piano *MASTER* sono l'oggetto della 2ª Revisione di progetto.

SF: responsabilità

Chi le scrive:

i sistemisti/analisti esperti sia dell'applicazione che del sistema in cui la applicazione va installata.

Chi le usa:

gli analisti/programmatori che realizzeranno il prodotto, i collaudatori per il disegno dei tests, i pianificatori, i technical writers, i manutentori, i revisori, il management.

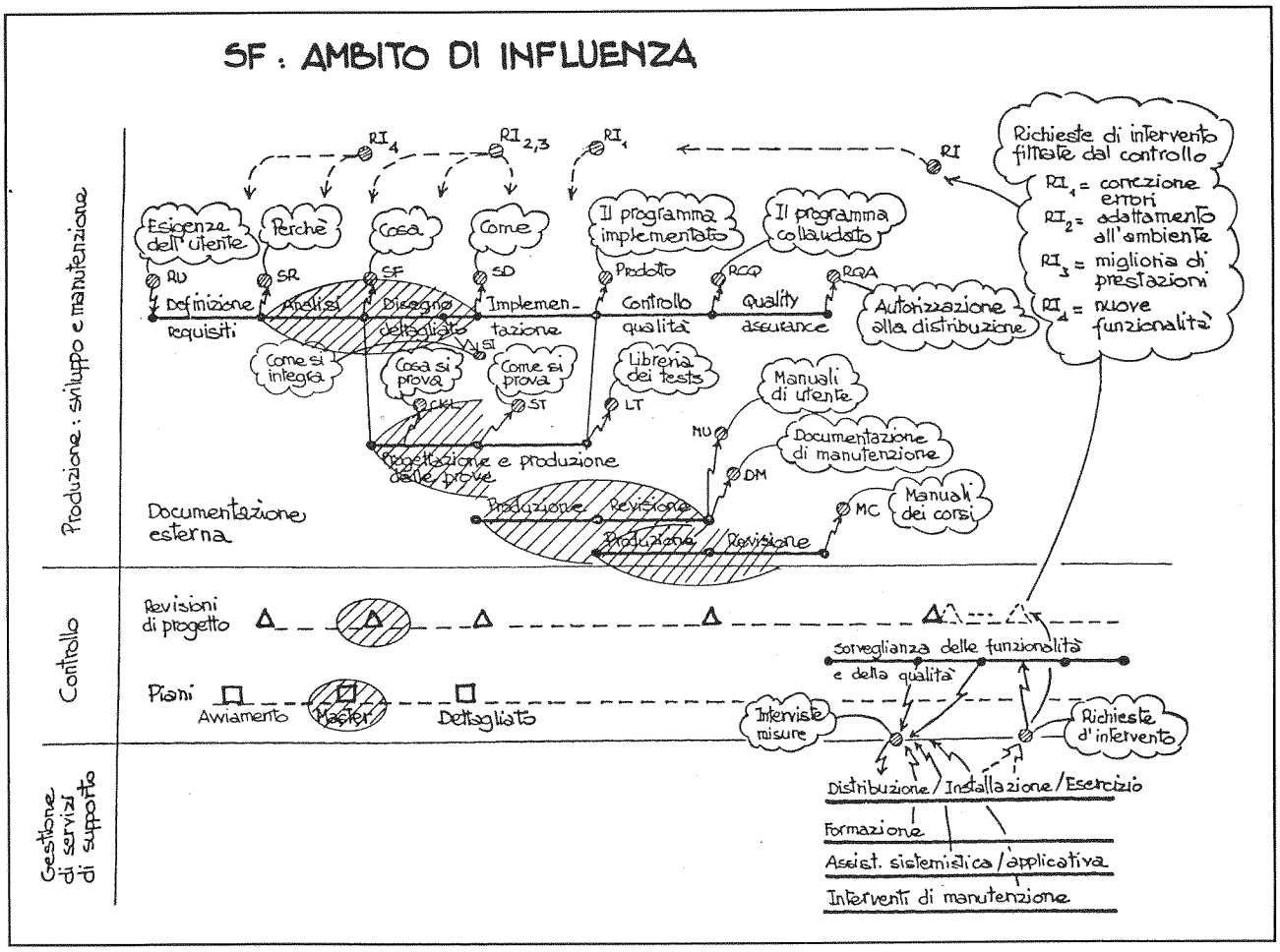


Fig. 15

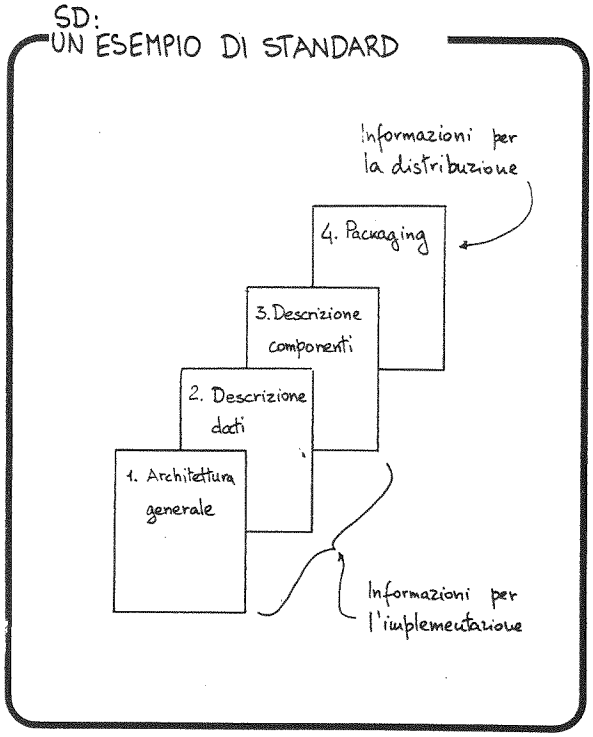


Fig. 16

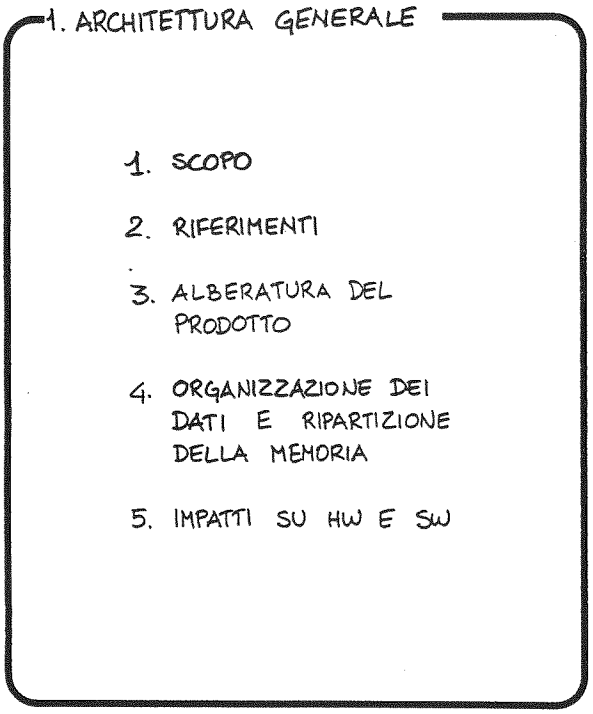


Fig. 16.1

4.6 Le Specifiche di Disegno (SD)

Esamineremo qui un esempio di standard di documentazione di specifiche di disegno, discutendo l'indice proposto e i contenuti di ciascun paragrafo. (fig. 16)

Osserviamo innanzitutto che il documento deve contenere:

- informazioni che permettano a chi deve sviluppare la costruzione di prodotti che si integrino correttamente
- informazioni che guidino gli aspetti di pianificazione dell'integrazione e di distribuzione.

1. Architettura generale

Il capitolo introduttivo, oltre che ricordare brevemente

- lo scopo del prodotto e
- la documentazione tecnica precedentemente prodotta,

deve fornire:

- l'alberatura del prodotto in termini di richiami tra i vari moduli, in modo da fornire un "indice" del programma,
- il "layout" della memoria durante l'esecuzione del programma (aree fisse, aree variabili, aree comuni, aree private, dimensioni, tipi, ecc.),
- l'elenco di nuove features hardware necessarie e l'elenco di nuove funzioni software che devono essere disponibili a supporto del prodotto. (fig. 16.1)

2. Descrizione dati

Allo scopo di permettere parallelismo nel lavoro, è necessario che tutte le interfacce tra i vari componenti siano definite con precisione, affinchè ciascun componente risulti completamente specificato e possa essere sviluppato senza rischi nella fase di integrazione.

Le interfacce vanno specificate a livello dei componenti principali, ovvero di quelli che vengono sviluppati in modo autonomo dagli altri.

La specifica di tali interfacce deve essere assolutamente completa con l'indicazione quindi, per ogni dato, di:

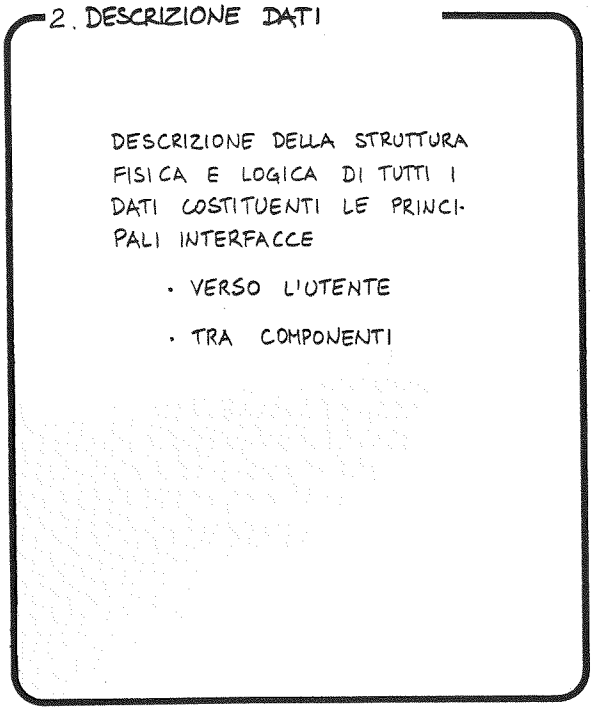


Fig. 16.2

Fig. 16.3

- nome
- tipo
- dimensioni
- valori possibili e significato
- moduli di lettura
- moduli di scrittura.

Tra le interfacce vanno considerati anche i dati utente, visti qui come ingressi o uscite specifiche di alcuni componenti. (fig. 16.2).

3. Descrizione componenti

Nell'ottica di un corretto sviluppo di programmi gerarchici per raffinamenti successivi, è opportuno che i vari componenti vengano definiti in modo preciso prima di una qualunque loro implementazione. (fig. 16.3)

Per ciascuno di essi è quindi necessario fornire le informazioni indicate, che permettono verifiche intuitive di consistenza, se non addirittura dimostrazioni di correttezza.

Nello sviluppo del singolo componente, operando rigorosamente con tecniche di programmazione strutturata top-down, è spesso impossibile definire tali informazioni se non durante la costruzione del programma stesso: lo standard di codifica illustrato in seguito riprodurrà lo stesso schema di informazioni.

4. Packaging

La consegna del prodotto avverrà come versamento, ad un opportuno servizio, di diversi tipi di componenti:

- documentazione
- programmi sorgenti
- moduli oggetto
- istruzioni di compilazione/linkaggio
- istruzioni di lancio
- .....

È necessario specificare, per ogni tipo di oggetto,

- nome
- tipo
- residenza su memoria di massa

Inoltre è necessario descrivere con dettaglio le procedure attraverso cui si ottiene il prodotto funzionante a partire da tutti i suoi componenti consegnati. (fig. 16.4)

SD: ambito di influenza

Le specifiche di Disegno sono il documento che guida in dettaglio tutta *l'attività* di *costruzione e integrazione del codice*, avendo specificato in anticipo i dati di I/O e l'elaborazione (PROCESS) di ogni componente.

Fig. 16.4

Informazioni su particolari funzionalità interne e dati dipendenti dal disegno possono essere usate per completare con *i casi interni la checklist.*

La conoscenza dettagliata di ogni componente consente di emettere un *piano dettagliato* che copre le singole attività fino alla distribuzione finale, piano che si integra (vedi il documento specifiche di Integrazione – SI) con il piano di Integrazione di una intera Release nel caso di progetti grandi.

Le SD, insieme con il piano dettagliato e le SI (quando queste sono applicabili) sono l'oggetto della 3ª revisione di progetto. (fig. 17)

SD: responsabilità

Chi le scrive:

gli analisti/programmatori che hanno a carico la realizzazione del prodotto.

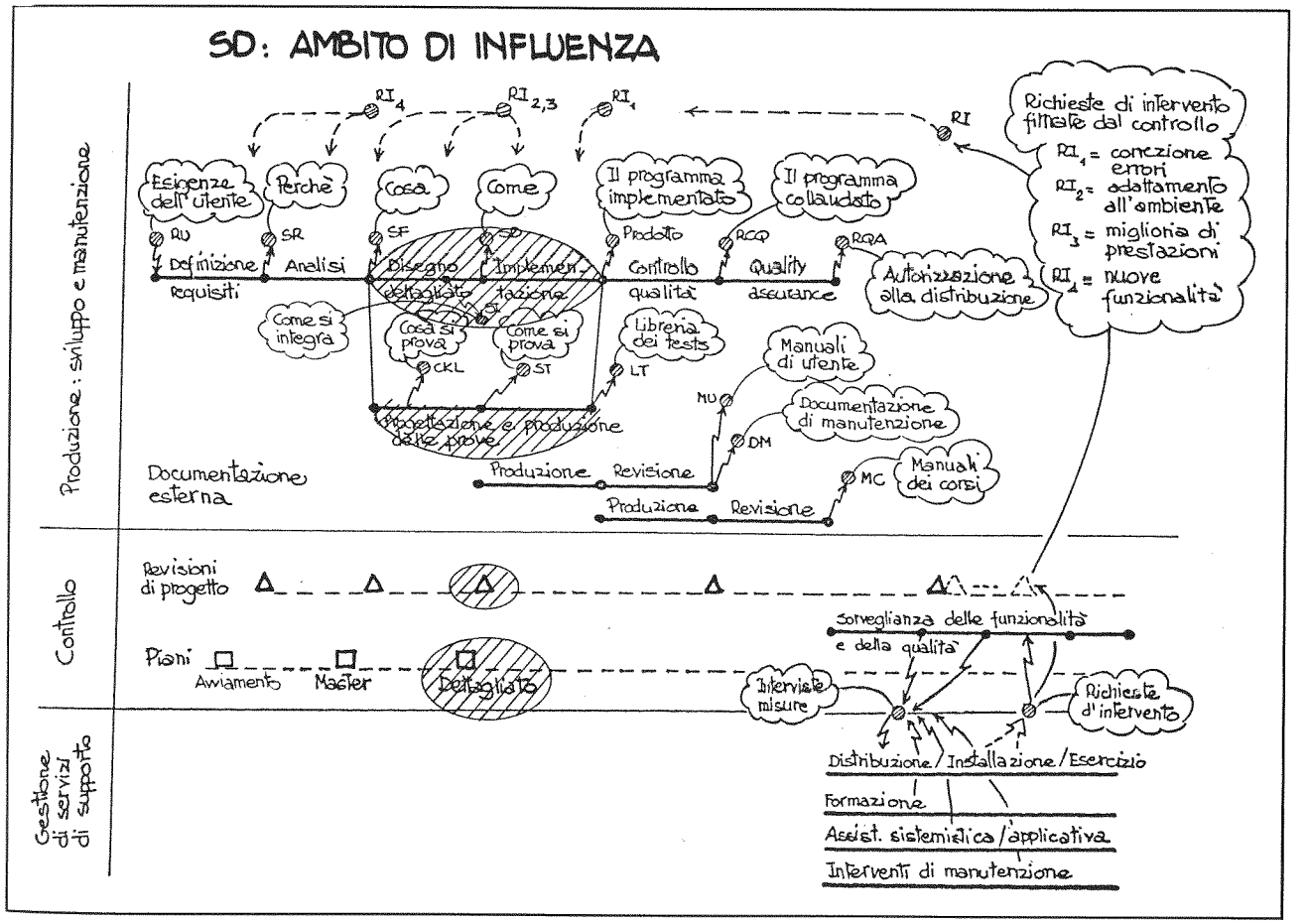


Fig. 17

Chi le usa:

gli stessi analisti/programmatori, i pianificatori, i collaudatori, i revisori di progetto, il management.

4.7 Le specifiche di Integrazione (SI)

Le Specifiche di Integrazione (SI) sono il documento che definisce la *precessione* delle operazioni necessarie per *integrare* un prodotto in un sistema più ampio.

Nei casi più frequenti – applicazioni semplici – il sistema supporta già il prodotto, ed è normalmente sufficiente eseguire le operazioni di compilazione, linkaggio, e inserimento in libreria del modulo caricabile e delle istruzioni di control language per poter usare il prodotto.
In tali casi le SI sono omesse: si usano le informazioni del paragrafo "Packaging" delle SD. (fig. 18)

Invece nelle applicazioni utente più complesse, costituite da molti programmi interdipendenti, e sempre nei casi di componenti di sistemi operativi, la sequenza di attività di "montaggio" (alias: integrazione) del sistema è da programmarsi in modo rigoroso e ragionato accuratamente, poichè alcuni componenti richiedono la disponibilità di altri già funzionanti per poter essere "debuggati" e collaudati, pena l'esigenza di ambienti di simulazione, in generale costosi.

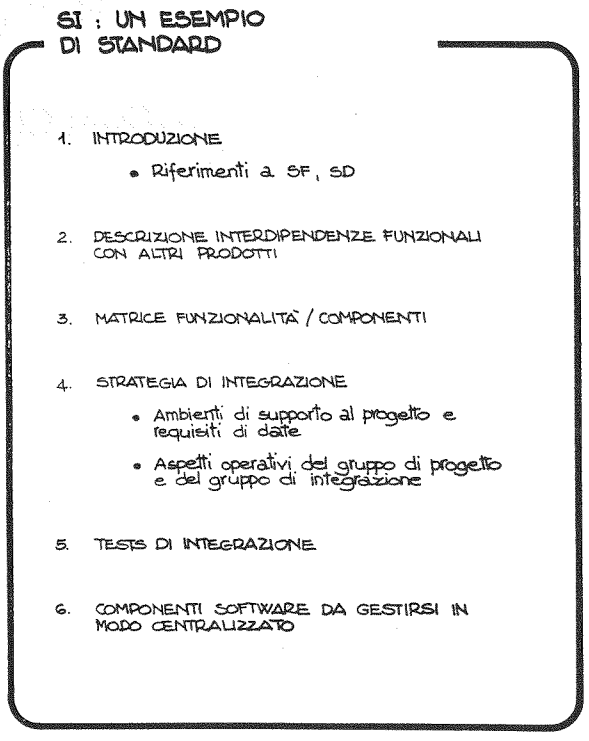


Fig. 18

Il documento tecnico che consente al gruppo di Integrazione di programmare al tempo richiesto la costruzione degli ambienti necessari ad ogni prodotto è appunto costituito dalle SI.
In esso i progettisti, con gli opportuni riferimenti alle SF e SD, descrivono le *interdipendenze funzionali* (le funzionalità che supportano il loro prodotto, e quelle da esso supportate in altri prodotti), precisano in una opportuna *matrice* quali sono i componenti supportatori/supportati per ogni funzionalità, e indicano per ogni funzionalità quali sono le *date di disponibilità volute* per rispettare gli obiettivi di piano. La stesura di queste informazioni va effettuata in modo coerente con la versione preliminare della strategia di Integrazione della intera applicazione o Release (il contenuto della Strategia di Integrazione di una Release è qui volutamente omesso – per esempio si veda (HISI 79 e METO 28)).
I progettisti devono anche specificare quali sono i *tests di integrazione* da costruirsi e da usarsi per ogni successivo ambiente in cui il prodotto in evoluzione via via sarà integrato, per verificarne appunto l'avvenuta integrazione. Infine si devono precisare quali sono i componenti di codice da *gestire in librerie centralizzate,* necessari alla costruzione dei suddetti ambienti da parte del gruppo di Integrazione.

SI: ambito di influenza

Le SI influenzano tutta la fase implementativa: esse sono l'esposizione di ciò che il progettista ha individuato essere la sequenza ottimale di "montaggio" nel sistema. (fig. 19)

Questa precessione di eventi (date di disponibilità di componenti) è la base per stendere un *piano di dettaglio ragionato e realistico*, in cui le attività si susseguono con sequenza tecnicamente corretta.

SI: responsabilità

Chi le scrive:

l'analista/programmatore responsabile della conduzione del progetto, (in collaborazione con il gruppo di Integrazione, ove applicabile).

Chi le usa:

il gruppo di progetto, i pianificatori, il gruppo di Integrazione, i revisori, il management.

4.8 La documentazione di Prodotto (DP)

Nell'ottica di una riduzione dei costi di gestione della documentazione e di miglioramento del modo di programmare è opportuno inserire nel sorgente del pro-

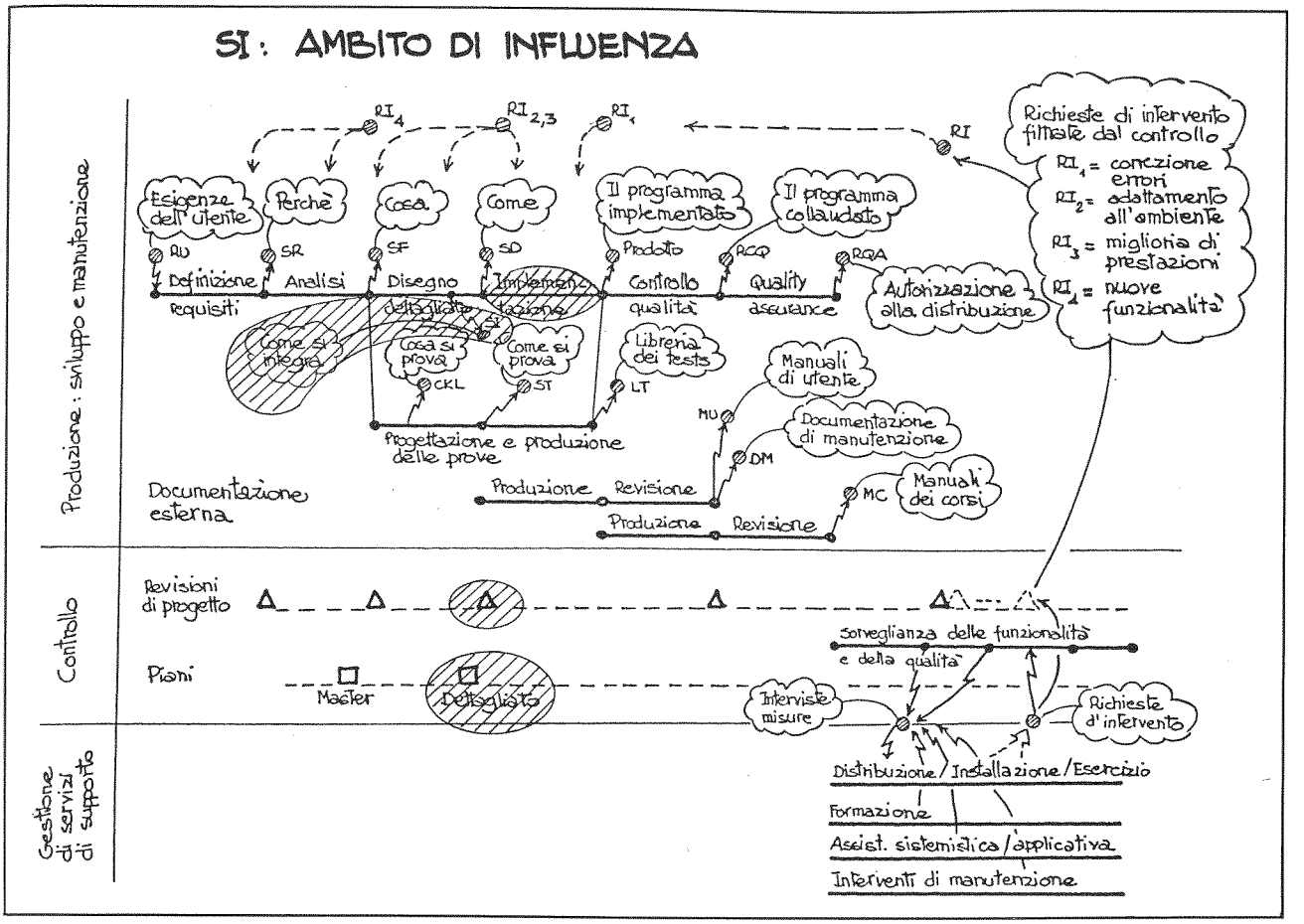


Fig. 19

gramma un insieme standard di commenti che lo documenti con completo dettaglio.
In figura sono indicati i cinque componenti fondamentali di tale standard di autodocumentazione. (fig. 20)

Si faccia riferiemento a "A. Chiapparino, A. Cicu, M. Maiocchi (CHIA 80) – A documentation method and the related tool for a software engineering environment, (AICA 1980) per approfondire in modo dettagliato gli aspetti di metodo e di strumenti necessari per l'autodocumentazione dei programmi. Il lavoro citato in particolare mette in luce vantaggi che si possono ricavare sull'attività di manutenzione.

Relazione tra SD e DP nei programmi autodocumentati.

Come si vede, con l'eccezione della testata, le voci che costituiscono una buona autodocumentazione dei

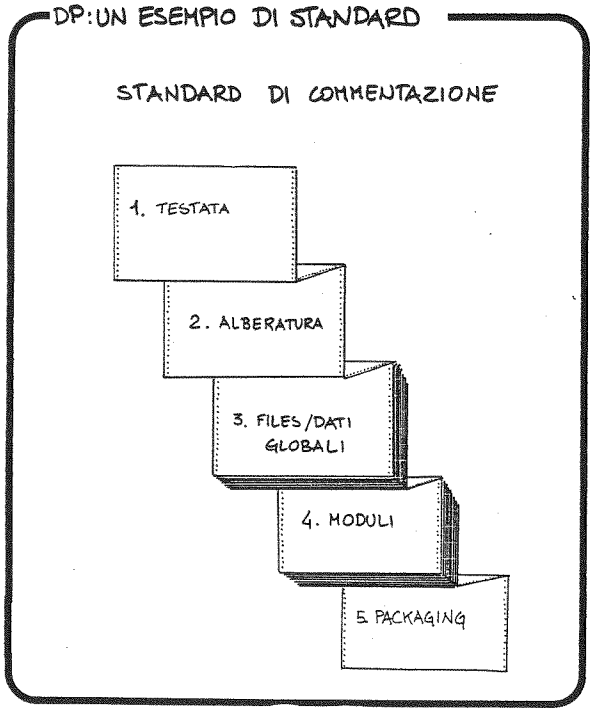


Fig. 20

programmi sono concidenti con le voci che compongono le specifiche di disegno. (fig. 21)

Non può che essere così, se le specifiche di disegno debbono essere specifiche che guidano effettivamente l'implementazione del codice dei vari componenti.

Ciò che si vuole sottolineare è che, nel caso che si realizzi effettivamente un codice autodocumentato (vedi ancora CHIA 80), per evitare un lavoro doppio è necessario scrivere le SD come prima stesura del codice sorgente dei componenti (in forma di blocchi di commenti): la disponibilità di un formattatore specializzato per la documentazione dei programmi estrarrà tali blocchi di commenti organizzando le SD con gli opportuni paragrafi (rif. CHIA 80).

Il documento DP risulta così alla fine essere costituito dal listing contenente le SD continuamente aggiornate e il codice definitivo dei componenti.

Non disponibilità di un formattatore

Nel caso che non sia disponibile un opportuno programma estrattore/formattatore, come conciliare la necessità di scrivere le SD complete pria di codificare (che l'assenza di un formattatore suggerirebbe di battere a macchina in un documento separato dal codice), con il vantaggio/esigenza di avere quelle informazioni nel codice?

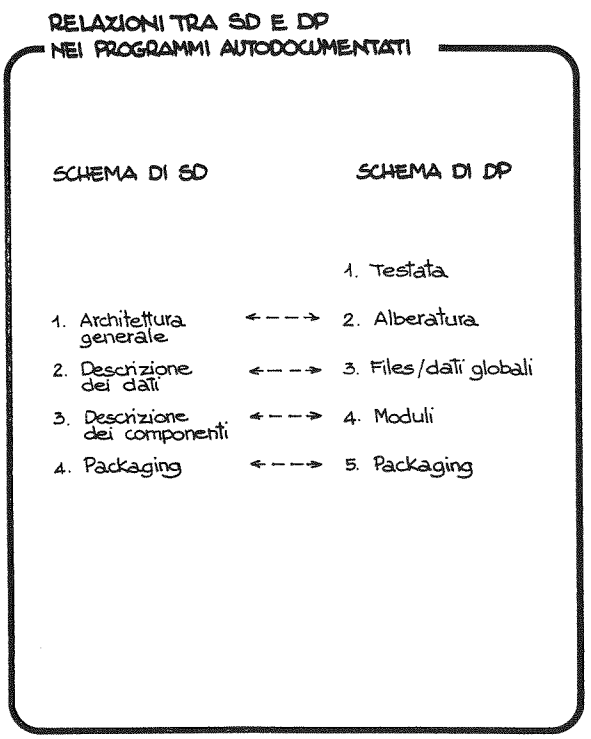


Fig. 21

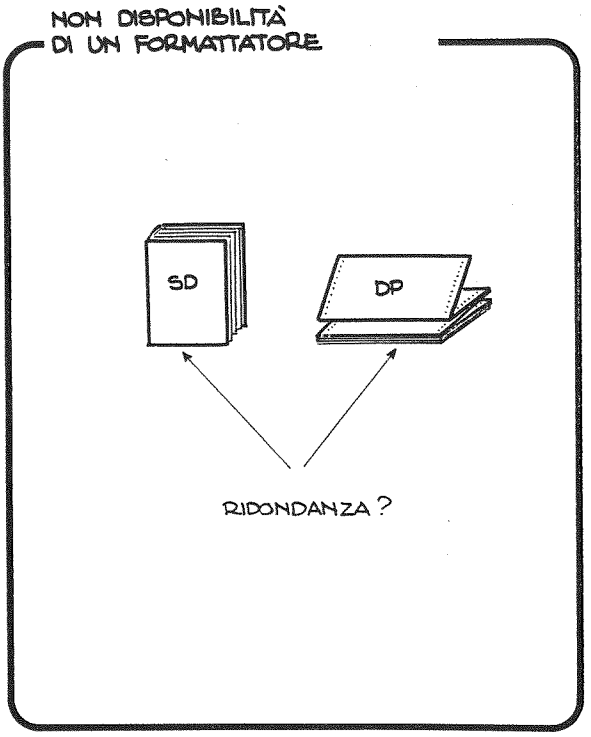


Fig. 22

Le soluzioni sono due (fig. 22):

1. SD battute a macchina e informazioni sui dati e moduli ripetute nel codice. Questa soluzione richiede una costosa manutenzione di due documenti separati, con il rischio che i due documenti ad un certo punto divergano come contenuto.

2. SD create fin dall'inizio come blocchi di commenti, ai quali poi si aggiunge il codice.
   Si ottiene lo svantaggio di avere un documento finale ampio (comprendente il codice), di consultazione un po' difficile (manca di indice, a causa dell'assenza del formattatore); e invece il vantaggio della manutenzione più facile di un unico testo (vedi ancora (CHIA 80)).

1. Testata

Pochi commenti, ove manchino frasi specifiche del linguaggio (presenti ad esempio in COBOL), permettendo di collocare completamente il programma nell'ambito del prodotto e dell'ambiente di produzione. (fig. 22.1)

Fig. 22.1

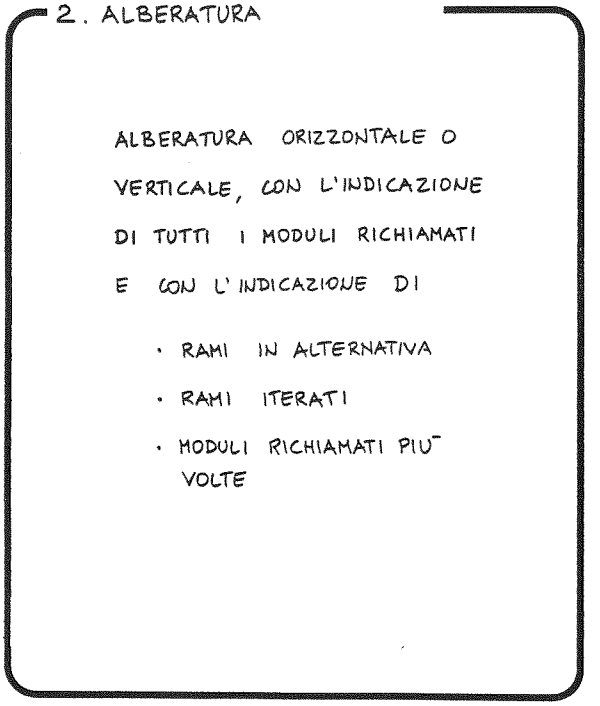


Fig. 22.2

2. Alberatura

Risultando il programma come collezione di moduli, è opportuno avere a disposizione un'alberatura che funga da indice: l'alberatura può indicare i nomi dei vari moduli richiamati e può fornire indicazioni sulle strutture (o sottoalberature) iterati o in alternativa.

È opportuna inoltre l'indicazione di quali moduli sono richiamati da più parti dell'albero, in modo tale che sia noto che una modifica su uno di essi debba essere sottoposta a verifica per tutti gli altri richiami. (fig. 22.2)

3. Files/Dati globali

Le indicazioni richieste valgono, come vedremo, anche per i dati locali che verranno descritti nei vari moduli. (fig. 22.3)

Si noti che spesso molte delle informazioni richieste sono già implicite nell'uso del linguaggio di programmazione: in tal caso, saranno ovviamente omesse.

4. Moduli

Lo standard proposto può costituire una vera e propria testata di commenti per ciascun modulo. (fig. 22.4)

Fig. 22.3

Fig. 22.4

Si intendono come aree di lavoro quelle aree usate dal modulo, ma assenti in ingresso e uscita: spesso si tratta di interfacce con moduli di livello più basso. Per queste, come eventuali dati locali disponibili (se ammessi dal linguaggio) si adottano le regole espresse al punto 3.

5. Packaging

Si veda il contenuto indicato nello standard SD.

DP: ambito di influenza

La Documentazione di Prodotto (DP) descrive come il prodotto è stato effettivamente realizzato. È costituita da una versione aggiornata delle SD, inserite o no nel corpo del codice sotto forma di blocchi di commenti.

La DP può essere usata come tale o con opportuni rifacimenti per produrre la *Documentazione di Manutenzione,* che è l'insieme dei documenti che supportano il personale di Assistenza Tecnica nel diagnosticare un errore Software (interventi di manutenzione).

Inoltre la DP è usata direttamente dai *manutentori* nello studio e nella realizzazione delle correzioni necessarie a fronte delle *Richieste di intervento.*

La DP è oggetto, insieme al resto della documentazione del prodotto, della revisione di qualità. (fig. 23, alla pagina seguente).

DP: respondabilità

Chi le scrive:

analisti/programmatori che realizzano il prodotto.

Chi le usa:

gli estensori della documentazione di manutenzione (spesso coincidono con gli autori della DP), i tecnici di assistenza, i manutentori, i revisori, il management.

4.9 La Checklist (CKL): struttura e obiettivi

La checklist è un documento avente *formato tabellare,* che deve consentire un facile *riferimento* (cross - reference) tra *le funzionalità del prodotto* da provare e i *nomi dei programmi di prova* disponibili o che si devono costruire per provare il prodotto stesso. (fig. 24)

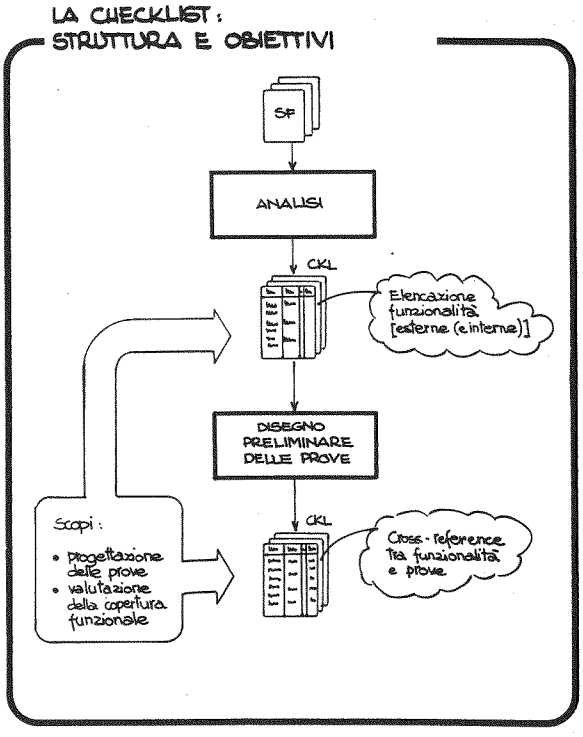


Fig. 24

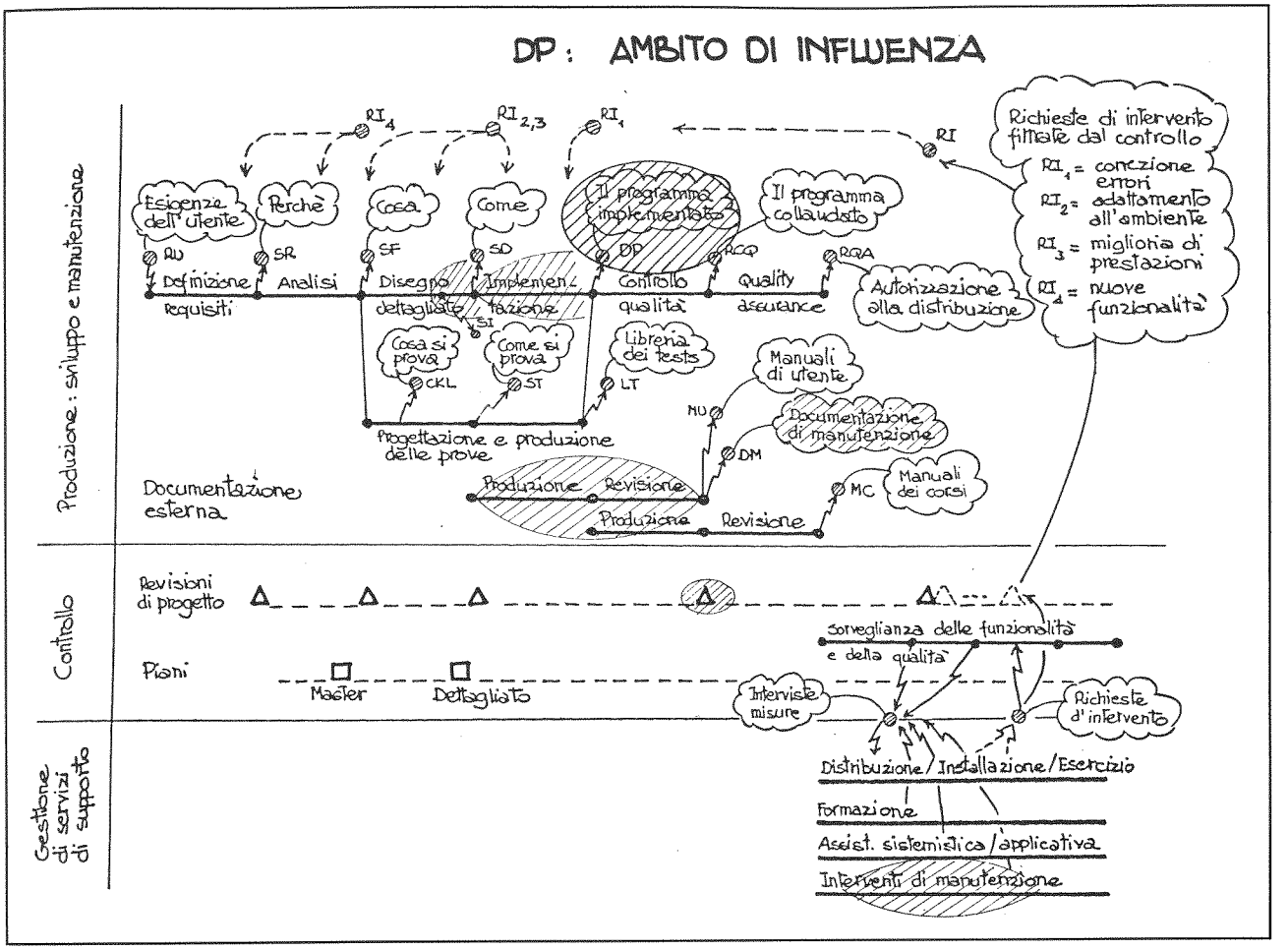


Fig. 23

La checklist è perciò anzitutto una *elencazione* in forma di lista di tutte le singole funzionalità distinte di un prodotto software, e di ogni altro aspetto non funzionale (per esempio le specifiche di prestazioni) che si intende provare; questa lista è prodotta attraverso una *analisi accurata del documento SF* che deve evidenziare ogni singola funzione esplicitamente menzionata ivi, ma anche le funzionalità implicite in esse (i test negativi ricadono spesso in questo caso – per es. come si comporta il prodotto se ad un parametro viene assegnato un valore non ammesso o non descritto nelle specifiche?).
Durante questa analisi è opportuno registrare nella CKL anche i dati di prova e la classificazione di priorità che si assegna ad ogni funzionalità.
La parte di checklist così ricavata dalle SF è chiamata *checklist esterna*, ed è la parte di gran lunga più importante, poichè un approccio di test ottimale deve essere guidato dalle funzioni (test funzionale). Qualora il progettista durante il disegno di dettaglio, abbia registrato i cammini e/o rami che dipendono dalla particolare soluzione implementativa scelta e i corrispondenti valori che i dati esterni devono assumere per percorrerli, queste informazioni debbono essere aggiunte alla checklist e ne costituiscono la parte cosiddetta di *checklist interna.*

La checklist così impostata persegue *due scopi:*

- supportare la *progettazione* dei programmi di prova
- consentire il continuo *controllo della copertura funzionale* delle prove stesse.

Il 2° passo infatti, nella compilazione della checklist, è dato dal disegno preliminare dei test, attraverso il quale si concepiscono i sigoli test e i raggruppamenti di funzionalità da esercitare con un singolo test, il cui nome è segnato sulla CKL a fianco di ogni funzionalità.
Si veda il lavoro (CERI 81) per una trattazione più ampia delle attività di analisi funzionale e progettazione dei tests.

La Checklist: quando?

Ci si potrebbe a ragione chiedere se non convenga usare il Manuale Utente (MU) invece delle SF, come base per l'analisi e la stesura delle checklits, dato che si vuole adottare un approccio funzionale al testing, ottenendo così il risultato di controllare anche il contenuto del manuale. (fig. 25)

I motivi per cui si ritiene migliore il partire dalle SF sono i seguenti:

. il MU generalmente è prodotto più tardi sul ciclo di progetto. Il suo uso per le checklists ne ritarderebbe la stesura. Si ritiene invece fondamentale per la solidità di un piano, che il disegno e implementazione dei tests avvenga in parallelo alla progettazione del prodotto (vedi fig. 6)

. l'uso delle SF per le checklist fa sì che le SF subiscano la migliore revisione possibile tempestivamente, quando ancora è possibile reagire a basso costo per una loro revisione e conseguenti modifiche o aggiunte sul disegno di dettaglio

. le SF includono aspetti (come le specifiche di prestazioni) che il MU non tratta o tratta solo in parte.

Il contrôllo di qualità del Manuale utente va affrontato in altro modo, come segue:

. una revisione a tavolino di esso, a fronte delle SF

. una redazione della parte descrittiva delle funzionalità contentute nelle SF, impostata in modo da poter essere trasportata tale e quale nel Manuale Utente, può

.. minimizzare il rischio di errori nel Manuale Utente

.. stimolare il progettista estensore delle SF a specificare il prodotto in termini più orientati all'utente, con gli innegabili vantaggi da ciò derivanti.

CKL: Un esempio di standard

Lo standard delle checklist può essere fornito come modulo prestampato; le varie colonne hanno il seguente significato (fig. 26):

. capitolo / paragrafo del documento di riferimento ovvero l'indicazione della collocazione della funzione (o elemento strutturale, o

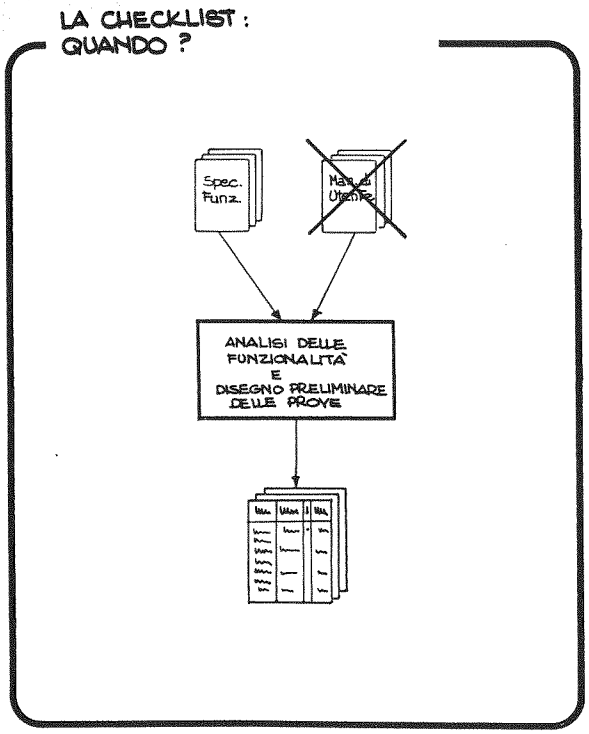


Fig. 25

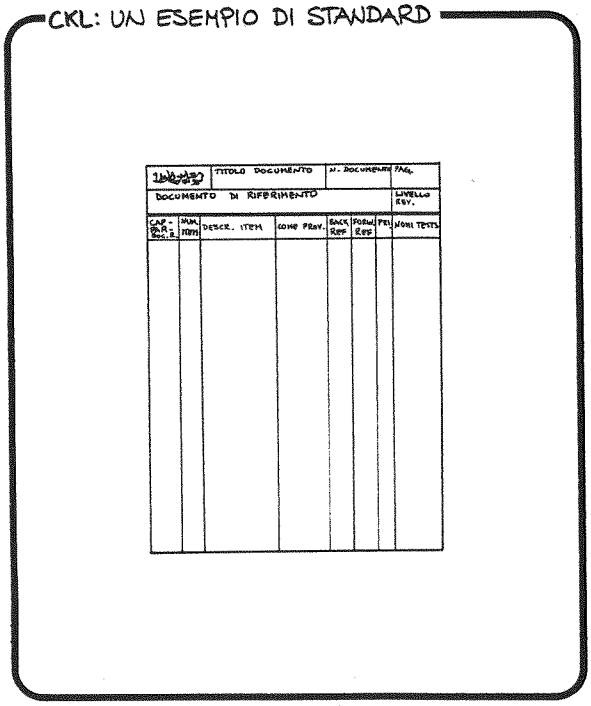


Fig. 26

funzionalità interna, in caso di ckl non funzionali) sul documento originale, in modo da permettere rapide e efficaci revisioni e/o manutenzioni

. n° item
ovvero n° identificativo della funzione e dell'elemento sotto controllo

. descrizione item
ovvero la trasposizione (possibilmente prelevata in modo completo dal documento) dell'elemento da controllare

. come provare
ovvero indicazioni sui dati di prova o sul modo di provare

. back/forw ref.
ovvero riferimenti ai documenti precedenti (da chi l'item è richiesto) (backword reference) e successivi (che cosa l'item comporta) (forword reference)

. priorità
livello di priorità da adottare nel controllo

. nomi tests
elenco dei nomi dei tests che esercitano l'item indicato.

CKL: ambito di influenza

La checklist supporta in modo determinante le seguenti attività:

. la progettazione e pianificazione delle prove

. la valutazione e il costante controllo delle copertura funzionale nelle fasi di

.. progettazione dei test
.. controllo di qualità
.. quality assurance

La valutazione della copertura delle prove consente inoltre di stimare la solidità e completezza del piano di dettaglio. I progettisti possono farne uso per la scelta di prove utili nella loro attività di debugging.

La CKL è oggetto della 3° revisione di progetto.

Infine la sorveglianza della qualità durante il periodo di esercizio fa costante riferimento alla CKL per comprendere più facilmente come gli errori rilevati possano essere sfuggiti alla attività di prova e decidere di conseguenza le eventuali estensioni delle libreria delle prove da parte dei manutentori. (fig. 27, alla pagina seguente).

CKL: responsabilità

Chi le scrive:

i collaudatori del prodotto

Chi le usa:

i collaudatori del prodotto, i progettisti, il gruppo di quality assurance, i revisori, il management, i manutentori.

Nota sul testing topologico

L'utilizzo di strumenti che consentono di registrare e conoscere, dopo l'esecuzione dei tests, quali rami di un programma sono stati eseguiti, va considerata come *integrazione del testing funzionale:* esso infatti consente di incrementare la libreria dei tests a basso costo, con tests "mirati" a coprire i rami lasciati scoperti dai tests funzionali realizzati.

Il solo testing topologico è insufficiente (oltre che essere troppo costoso se affrontato come approccio unico): infatti anche il 100% di copertura topologica non ci direbbe mai ad es. se ci si è dimenticati di implementare una certa funzionalità; solo l'analisi funzionale lo può rilevare.

4.10. Le Specifiche di Test (ST)

Questo documento oltre a fornire elementi di collocamento dei tests, deve provvedere:

. al raggruppamento dei tests in unità di lancio (catene)

. alla desrizione dei singoli tests

Il documento è conseguente all'ottenimento completo delle checklists, ovvero alla completa definizione di quanti e quali tests dovranno essere fatti, ed ha lo scopo di definire con precisione quello che potremo considerare "le specifiche funzionali del prodotto di supporto 'test'": esso è quindi orientato alla costruzione ed alla manutenzione dei tests. (fig. 28)

1. Introduzione

Oltre ad indicazioni concise sullo scopo del test ed a riferimenti, è necessario indicare (fig. 28.1):

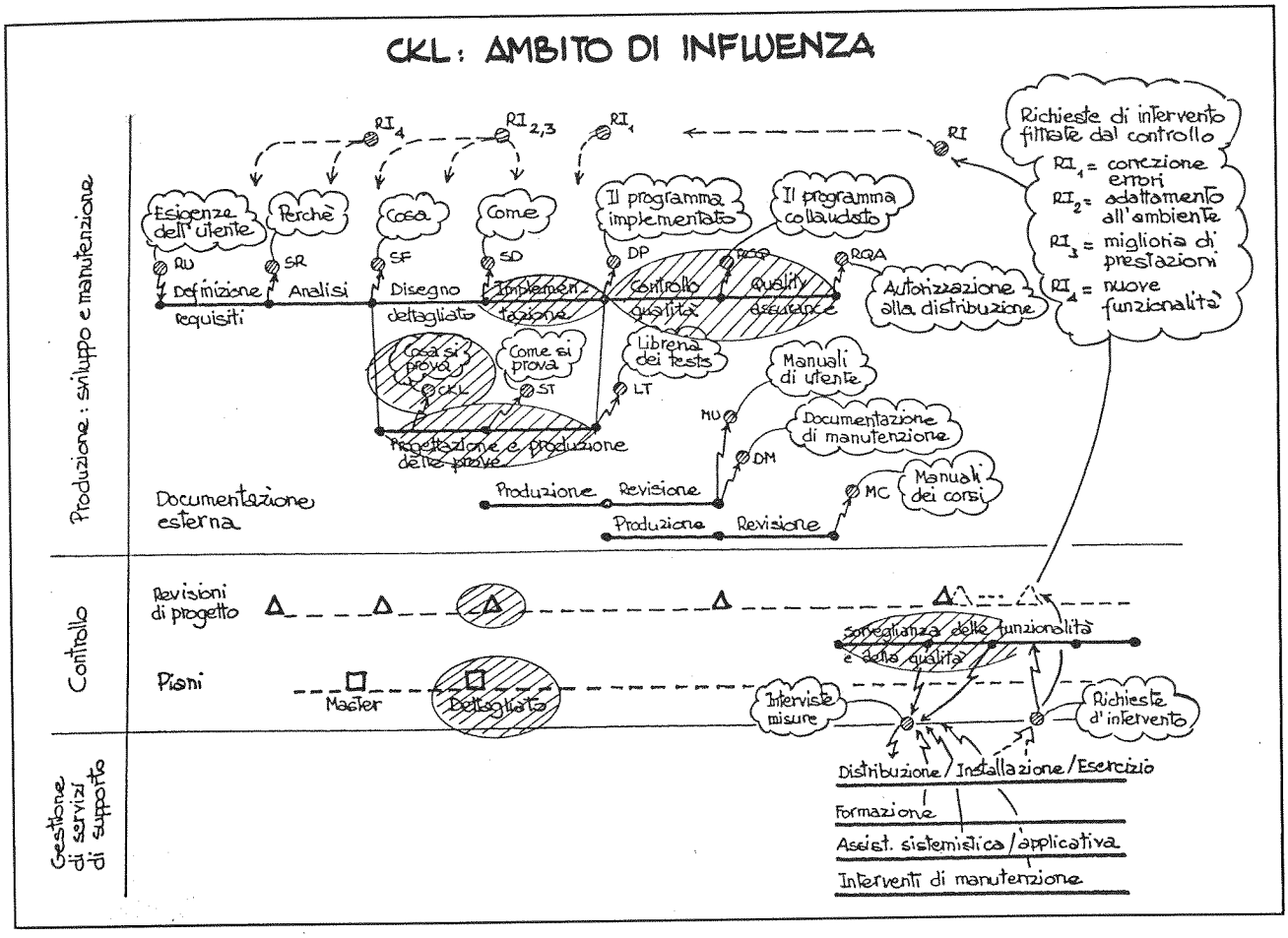


Fig. 27

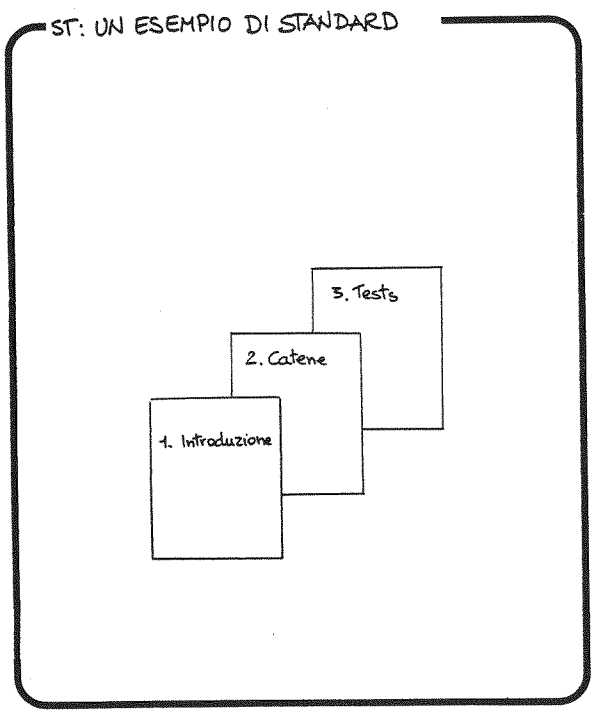


Fig. 28

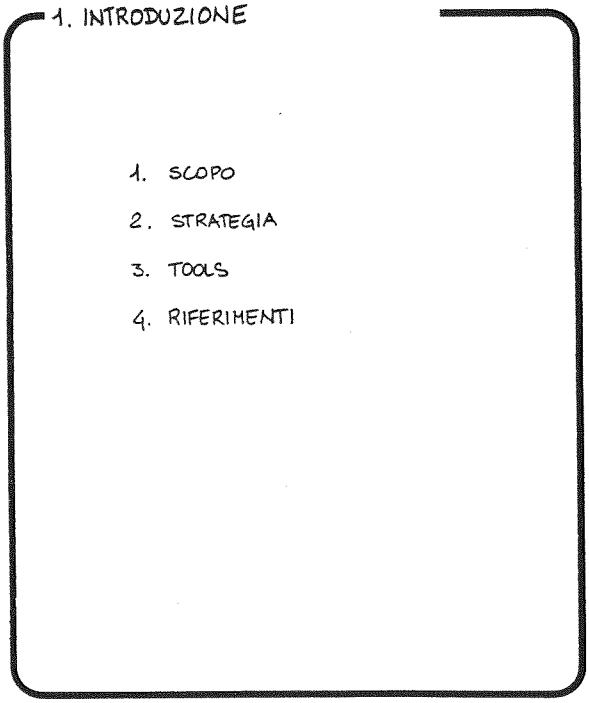


Fig. 28.1

. la strategia di testing, ovvero indicazioni dei criteri (funzionali/topologici), dei metodi di controllo (automatico/visivo) e delle fasi di controllo (se necessario)

. i tools impiegati, (eventualmente anche sviluppati) per l'esercizio del testing.

Strategia: Esempio

A titolo di semplice esempio di strategia di test, esaminiamo brevemente il caso di prove su programmi di ordinamento. (fig. 28.1.1)

La verifica di corretto funzionamento di programmi di sort, benchè molto semplice da un punto di vista concettuale, risulta estremamente complessa a causa di elementi, tra cui, fondamentalmente:

. l'esigenza di costruzione di files di dimensioni non banali, adeguati al problema, con records opportunamente disordinati

. l'esigenza di effettuare controlli sui risultati, che, se fatti mediante controllo visivo, rischiano di essere eccessivamente costosi e a scarsa garanzia di correttezza del controllo:
si pensi ad esempio alla verifica che i records siano ordinato, che nessun record sia perso, che nessun record sia stato inserito, che il contenuto di ciascun record non sia stato alterato.

Per risolvere il problema...

Strategia: Esempio

...si può adottare la seguente strategia, da dichiarare e motivare nel documento in questione (fig. 28.1.2):

. adozione di un Test Data Generator, ovvero di un programma che, accettando in ingresso descrizioni compatte di files, ne generi campioni, anche di grandi dimensioni, con il "disordine" necessario, da fornire come ingresso al sort, e i files equivalenti già ordinati (ovvero contenenti gli stessi records nell'ordine che ci si attende dopo l'esecuzione del SORT)

. adozione di un comparatore di files (costruito ad hoc, ovvero preso dalla letteratura), che permetta di automatizzare il controllo del risultato, fornen-

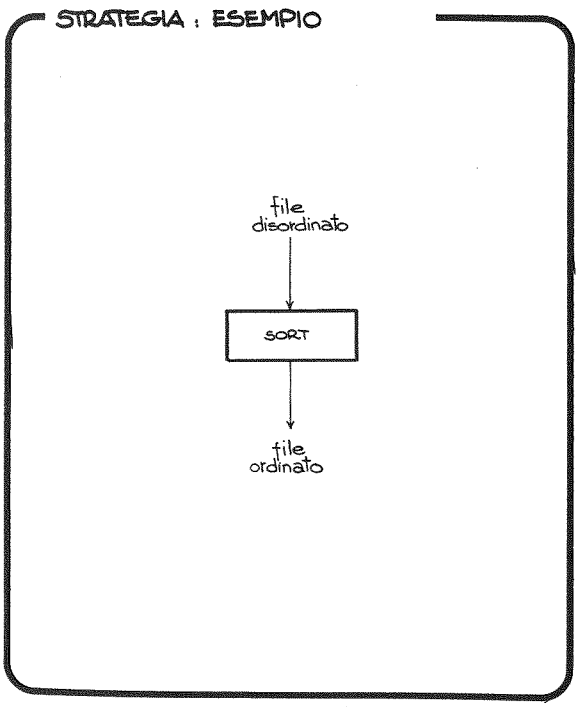


Fig. 28.1.1

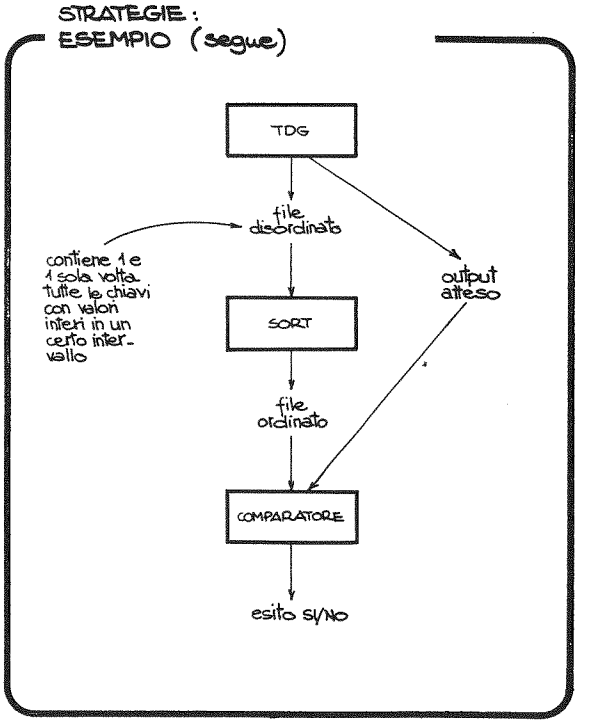


Fig. 28.1.2

do informazioni booleane (test andato a buon fine/esito negativo) e informazioni aggiuntive adeguate al debugging nel caso di esito negativo.

2. Catene

Il raggruppamento dei tests in catene ha lo scopo di fornire unità di lancio più grandi del singolo test, e quindi di semplificare l'esercizio del testing.

Per la loro descrizione può essere adottato un modulo come quello riportato: si noti che l'elenco delle funzioni testate può essere costituito dall'elenco dei numeri ricavati dalle checklists.

Si noti inoltre che il modulo può essere usato anche per una verifica dello stato di sviluppo, nelle ultime colonne (compilato/linkato sono usati nel caso di programmi a fronte di un controllo automatico). (fig. 28.2)

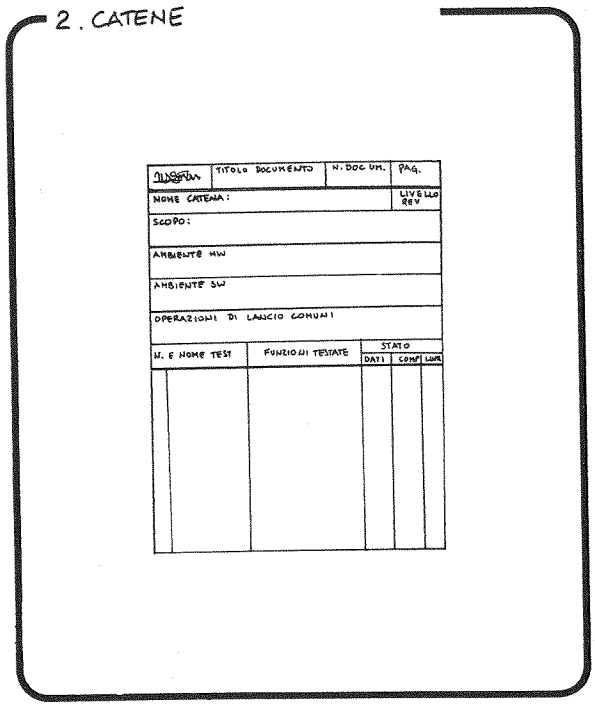


Fig. 28.2

3. Tests

Anche per la descrizione dei singolo test può essere adottato convenientemente un modulo come quello illustrato. (fig. 28.3)

In particolare si osservi il campo "RISULTATI":

esso richiede la definizine di due tipi diversi di risultati, forniti da prodotti diversi:

- il primo, che nel modulo è indicato con "RISULTATI ATTESI", fa riferimento all'output atteso dal programma sotto test: è l'uscita fornita dalle "FUNZIONI ESERCITATE" a fronte dei "DATI DI INGRESSO"

- il secondo, indicato come "MSG SUCCESSO / MSG FALLIMENTO" è il risultato dell'eventuale controllore automatico a fronte della verifica che i risultati attuali della esecuzione corrispondano ai risultati compione preparati coerentemente con la specifica dei "RISULTATI ATTESI".

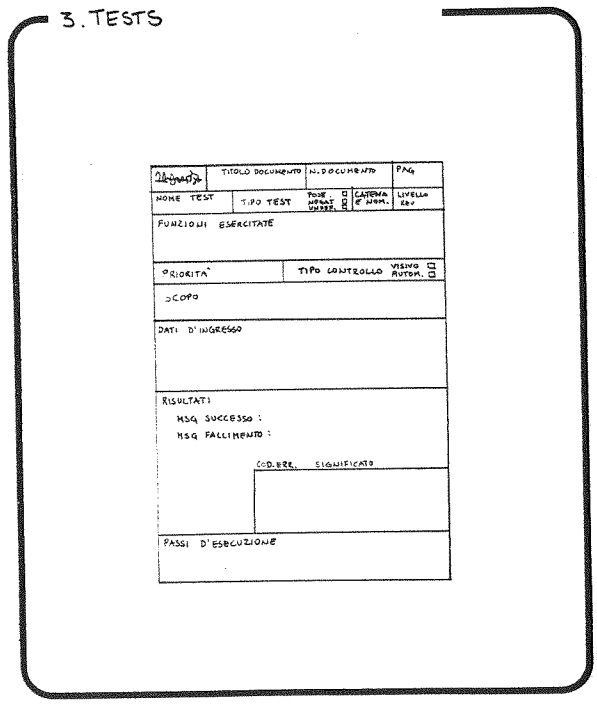


Fig. 28.3

ST: ambito di influenza

Le specifiche di test sono per i programmi di prova l'equivalente delle SD per il prodotto (fig. 29): esse ne consentono una realizzazione ottimale che concilii le esigenze contrastanti di:

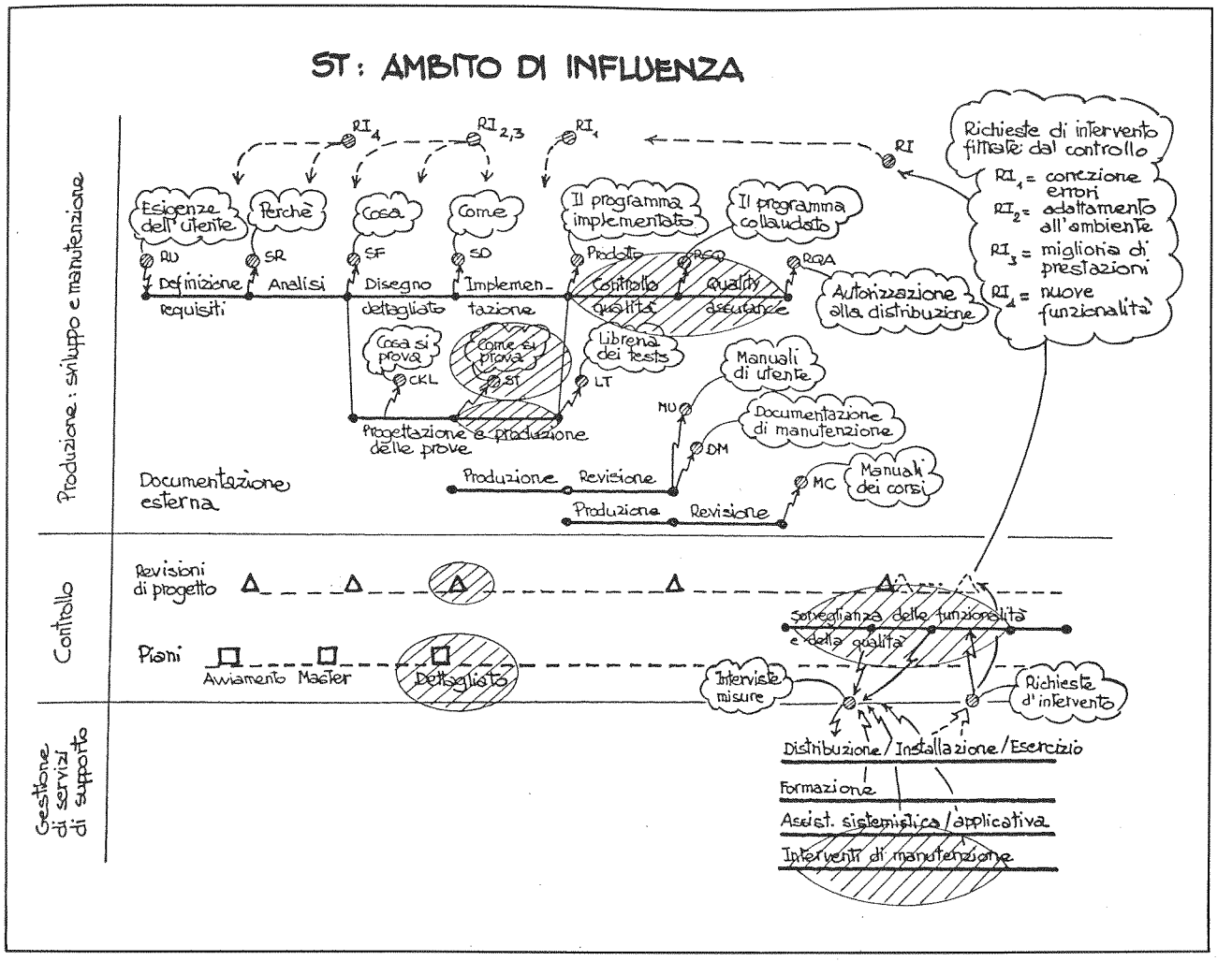


Fig. 29

- massimo potere risolutivo del singolo test (cioè minimo numero di funzionalità esercitata per test, e quindi molti test)
- minimo costo di esercizio (perciò poche unità distinte lanciabili e controlli automatici dei risultati)
- strategia di esercizio dei tests modulare e modificabile facilmente a fronte di errori che bloccano una parte della libreria dei tests.

Perciò un buon disegno di test che realizzi i detti obiettivi è la chiave per avere non solo la massima produttività del testing nelle fasi di Quality Control e di Quality Assurance, ma anche la capacità di un rapito ricollaudo delle correzioni apportate per gli interventi di correzione.

Le ST, oltre che avere gli aspetti I/P/O tipici delle SD, devono inoltre contenere gli aspetti di uso dei tests stessi (analogia con SF), ad uso degli operatori del centro chiuso. Le ST sono oggetto della 3° revisione di progetto e contribuiscono alla stesura del piano dettagliato.

ST: responsabilità

Chi le scrive:

i collaudatori.

Chi le usa:

i collaudatori, gli operatori del centro chiuso, i manutentori, i pianificatori, i revisori, il management.

4.11 Il rapporto di controllo qualità

I diversi elementi che devono costituire il rapporto del controllo qualità sono (fig. 30):

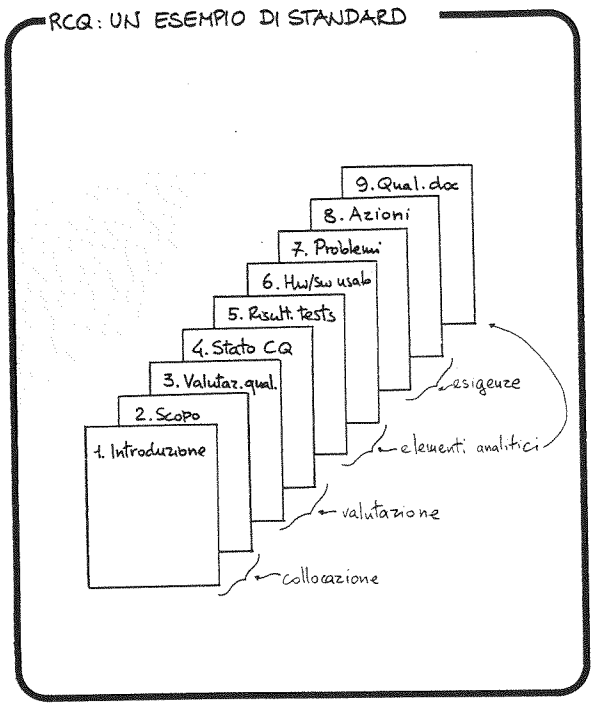


Fig. 30

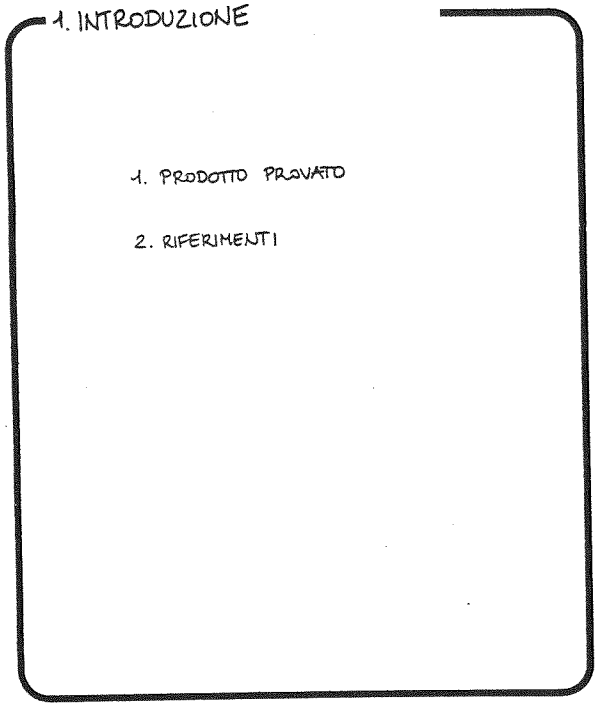


Fig. 30.1

- elementi di collocazione e inquadramento che definiscano ambito e limiti del controllo
- elementi di valutazione che indichino la qualità misurata del prodotto.
- elementi analitici che giustificano la valutazione precedente
- elementi informativi su esigenze sentite per migliorare la qualità del prodotto.

1. Introduzione

L'introduzione può limitarsi ad una indicazione del prodotto a cui si fa riferimento, con relativa indicazione di altri documenti di sviluppo. (fig. 30.1)

2. Scopo

Questo capitolo richiama brevemente la strategia di testing già specificata nelle ST. (fig. 30.2)

3. Valutazione della qualità

Questo capitolo è chiaramente l'elemento centrale del documento e deve fornire indicazioni quantitative e qualitative su ciascuno degli argomenti indicati. (fig. 30.3)

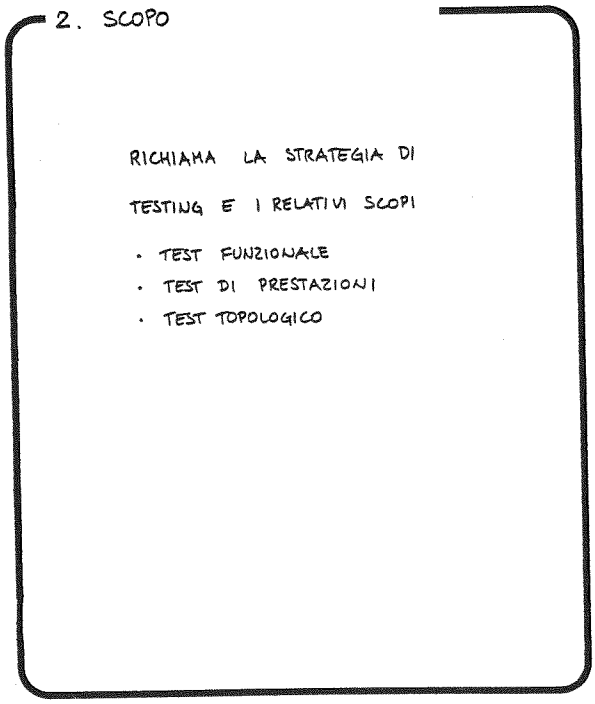


Fig. 30.2

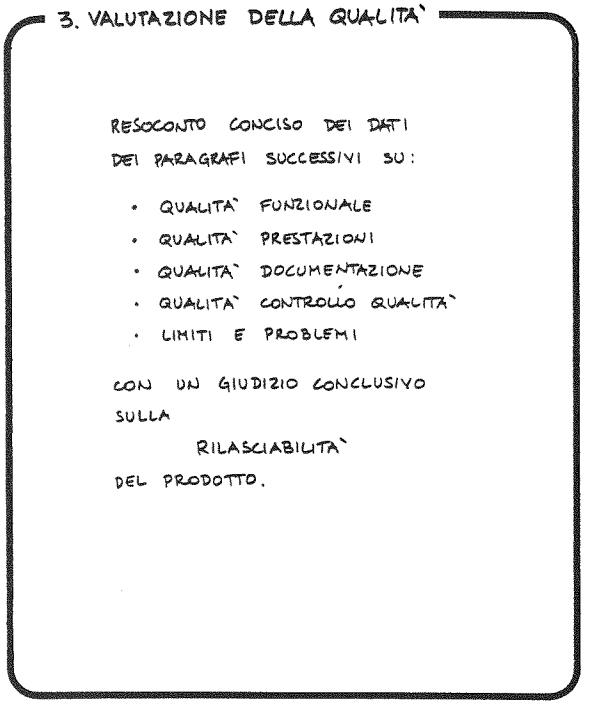


Fig. 30.3

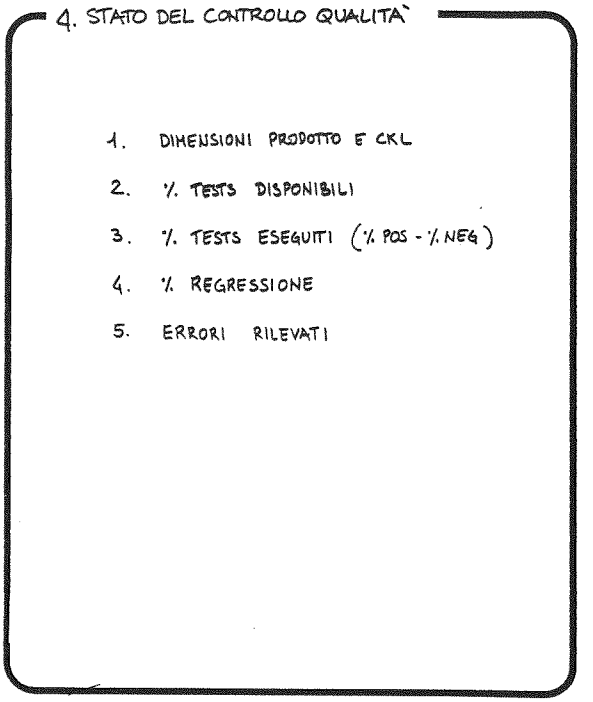


Fig. 30.4

In particolare deve fornire resoconto su

. n° di funzioni esercitate con esito positivo e n° con esito negativo (quantitativo)

. riscontro sulle prestazioni (quantitativo)

. adeguatezza della documentazione (qualitativo)

. adeguatezza del controllo qualità (quantitativo) ovvero il rapporto tra la dimensione fisica del documento delle ckl (ad esempio in termini di n° di funzioni) e la dimensione fisica delle specifiche del prodotto (ad esempio in termini di n° pagine) o del prodotto sviluppato (ad esempio in termini di n° di linee di codice), confrontando con altri casi desunti da esperienze precedenti.

. limiti del prodotto

La rilasciabilità è giudicata indicando qualitativamente il grado di confidenza del prodotto e analiticamente l'esistenza di possibili "bypass" o difficoltà d'uso per ogni limitazione.

4. Stato del controllo qualità

L'elenco delle informazioni richieste fornisce un quadro completo di (fig. 30.4):

. testing presumibilmente necessario

. testing predisposto

. testing effettuato

. esiti

5. Risultati dei tests

I risultati analitici possono essere convenientemente raccolti in moduli come quello indicato (fig. 30.5).

È possibile organizzare il controllo qualità e la costruzione dei tests con opportuni strumenti che permettano la gestione automatica e la stampa dei reports degli esiti delle prove: si veda a tale proposito il lavoro "Farnedi S., Poggiali C., Polillo R., Vignoni A.: TRP: un sistema per la gestione dei risultatil delle prove di un sistema operativo – A.I.C.A., 1976 Milano". (FARN 76).

6. HW / SW usato

Vengono definiti con dettaglio le caratteristiche delle configurazioni hw impiegate nel testing e le versioni e configurazione dei sistemi sw relativi. (fig. 30.6)

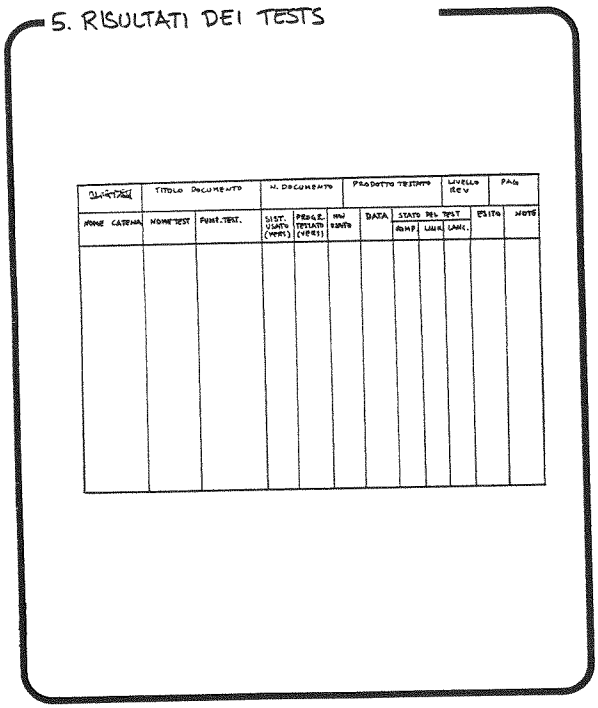


Fig. 30.5

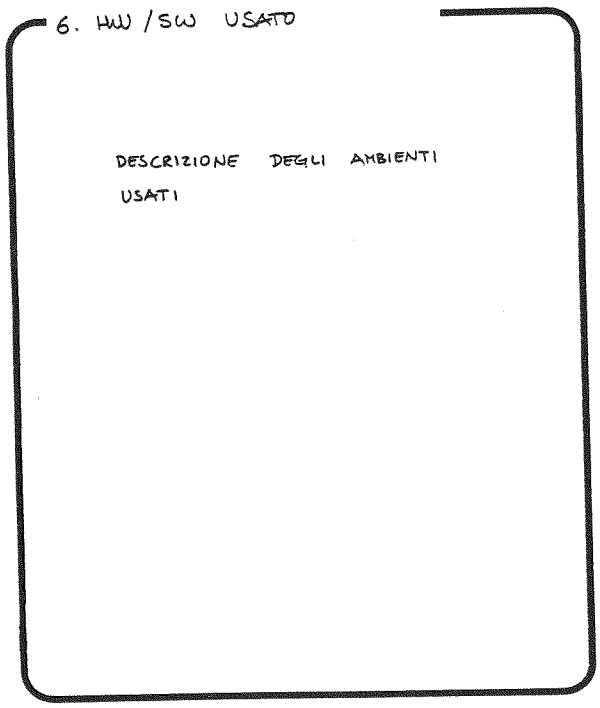


Fig. 30.6

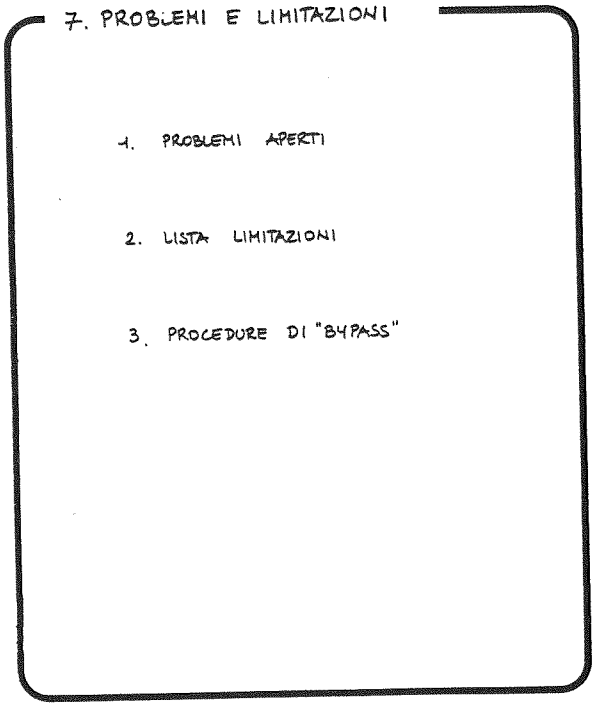


Fig. 30.7

7. Problemi e limitazioni

La lista indicata di argomenti deve essere sviluppata dettagliando le condizioni di malfunzionamento, le presunte cause, le possibili procedure per ovviare al problema dal punto di vista dell'utente. (fig. 30.7)

8. Azioni

Il capitolo deve elencare tutte le azioni che

. devono assolutamente
. è opportuno che siano
. potrebbero essere

intraprese per

. aumentare la confidenza del prodotto
. migliorare la qualità
. eliminare limitazioni
. permettere il rilascio (fig. 30.8)

9. Qualità della documentazione

La qualità di un prodotto è completa solo a fronte di una qualità adeguata della sua documentazione. (fig. 30.9)

Questo capitolo ha lo scopo di fornire le indicazioni analitiche su quali controllo sia stato effettuato sulla documentazione e sui relativi risultati.

Eventuali problemi/azioni vengono elencati nei capitoli 7 e 8.

Un resoconto conciso è fornito nel capitolo 3.

RCQ: ambito di influenza

Il Rapporto di Controllo Qualità fornisce gli elementi di guida delle attività finali di testing e di revisione dei documenti:

Inoltre esso definisce

. le azioni correttive da intraprendere nella fase di Quality Assurance, prima della distribuzione del prodotto.

. le informazioni sulle limitazioni che dovranno essere note all'utente e al personale di assistenza tecnica e sistemistico/applicativa per il corretto uso del prodotto stesso.

La valutazione analitica della qualità e delle limitazioni, effettuata dal gruppo di progetto e contenuta nel RCQ, è la base per una valutazione indipendente della qualità fatta dal gruppo Quality Assurance, il quale deve fornire al management un parere tecnico sulla qualità che sia indipendente da quello dei progettisti.

Il RCQ è oggetto della revisione finale di Qualità, insieme al resto della documentazione del prodotto. (fig. 31, alla pagina seguente).

RCQ: responsabilità

Chi le scrive:

il responsabile del gruppo di progetto oppure il responsabile del collaudo (se persona diversa dalla precedente)

Chi le usa:

il gruppo di Quality Assurance, i revisori, il management.

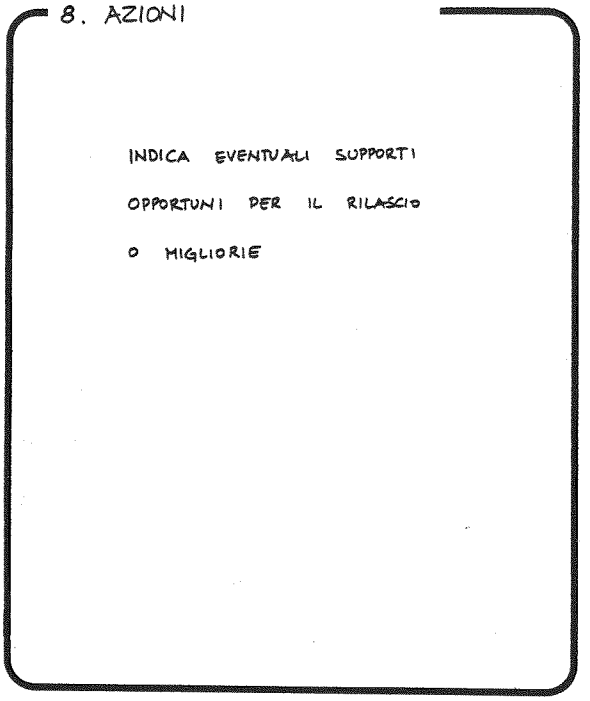


Fig. 30.8

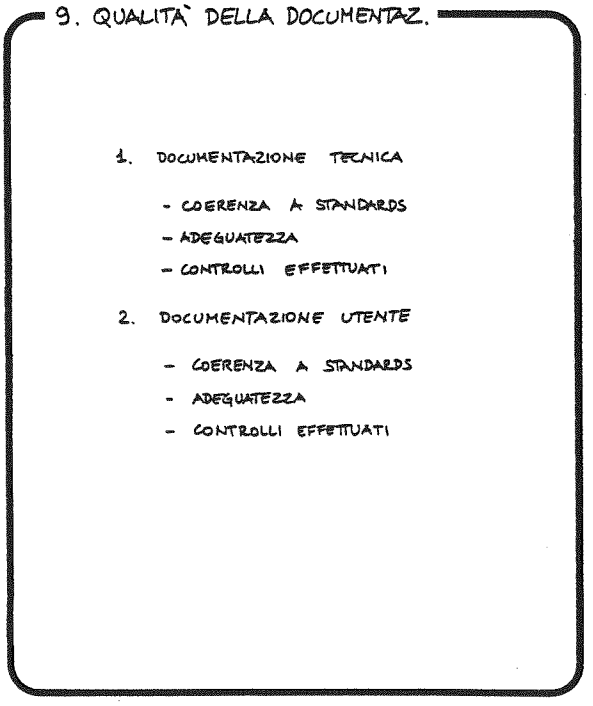


Fig. 30.9

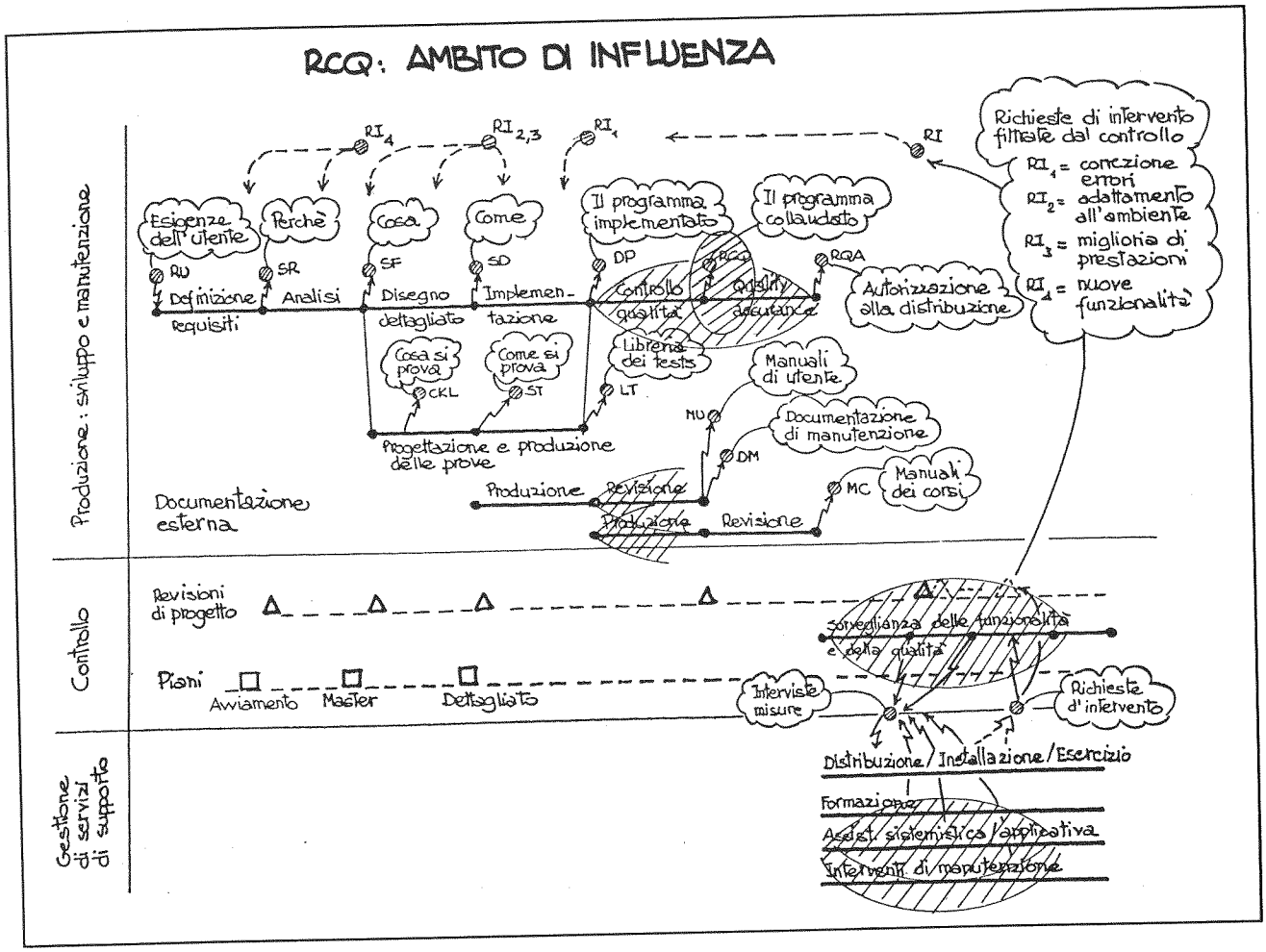


Fig. 31

BIBLIOGRAFIA

- AIEL 80 – L. Aiello, G. Prini – *L'automazione degli uffici: una panoramica* – Rivista di Informatica – vol. X, n° 4 – Ottobre/Dicembre, 1980
- ANS 79 – *Full Report of the flowchart committee on ANS Standard x3.5* – 1970 – SIGPLAN Notices (ACM) – vol. 14 n° 3 – Marzo 1979 – 16-27
- BALZ 78 – R. Balzer, N. Goldman, D. Wile – *Informality in program specifications* – IEEE Transaction on Software Engineering – vol. SE-4 n° 2 – Marzo 1978 – 94-103
- BAUE 75 – F.L. Bauer (ed.) – *Software Engineering: an advanced course* – Lecture Notes in Computer Science – Springer Verlag 1975
- BATE 81 – J.P. Bateman, S.A. Gloss Soler – *Developing a Standard for Software Engineering Terminology* – nei Proceedings di (SWEN 81)
- BELA 76 – L.A. Belady, M.M. Lehman – *A model of large program development* – IBM Systems Journal – n. 3 – 1976
- BELA 80 – L.A. Belady, C.J. Evangelisti, L.R. Power – *GREENPRINT: a graphic representation of structured programs* – IBM Systems Journal, vol. 19, n. 4 – 1980
- BELF 76 – P.C Belford, A.F. Bond, D.G. Menderson, L.S. Sellers – *Specifications: a key to effective software development* – 2nd International Conference on Software Engineering (13-15 October 1976 – S. Francisco, Ca, – IEEE cat. – 76 CH 1125 – 4C)
- BERR 75 – D.M. Berry – *Structured Documentation* – SIGPLAN Notices of ACM – Nov. 1975
- BERS 78 – E.H. Bersoff, V.D. Menderson, S.G. Siegel – *Software Configuration Management* – Proc. of SW Quality Assurance Workshop – Nov. 15-17, 1978 – (SEN vol. 3 n° 5 ACM)

- BETT 77 – T. Bettinazzi, M. Maiocchi, A. Maserati, F. Monzani, E. Spoletini E. Toscani – *PHOS: una metodologia per la costruzione veloce e affidabile di programmi* – Note di Software n. 4, 1977

- BOEHM 76 – Barry H. Boehm – *Software Engineering* – IEEE Transactions on Computers, vol. C–25, n° 12 – December 1976

- BOLO 81 – S. Bologna, W. Eherenberger, J.R. Taylor, U. Voges – *Prestandardisation activities for computer based Safety Systems* – nei Proceedings di (SWEN 81)

- BOWE 79 – J.B. Bowen – *A Survey of Standards and proposed metrics for Software Quality Testing* – Computer – Agosto 1979

- BROC 81 – J. Brockman – *Teaching Computer Documentation in an Academic Setting* – Asterisk vol. 7 n° 4 – July 1981 (ACM SigDoc)

- BROO 78 – R. Brooks – *Using a Behavioral Theory of Program Comprehension in Software Engineering* – (3rd International Conference on SW Engineering – May 10/12, 1978 – Atlanta – IEEE cat. 78 CH 1317 – 7C)

- BROO 81 – R. Brooks (Chairperson) – vedi tutta la sezione M – 3 su *"Human Engineering"* dedicata alla comprensione dei programmi – Proceeding of 5th Internationa Conference on Software Engineering, March 9/12, 1981 – S. Diego, Ca – pagg. 210-230

- BROW 74 – *Programming and documenting software projects* – ACM Computing Surveys – vol. 6 n. 4 December 1974

- BUCK 78 – J.K. Buckley, J.T. Pedersen – *Kongsberg road to an Industrial Software Methodology* – (3rd International Conference on Software Engineering – May 10/12, 1978 – Atlanta – IEEE cat. 78 CH 1317 – 7C)

- BUCK 79 – F. Buckley – *A standard for Software Quality Assurance Plans-Computer, August 1979*

- BUCK 81 – F. Buckley – *A standard for software quality assurance plans* – A status report. p. 87 in (SWEN - 81)

- CAPP 77 – S. Cappelli, V.De Antonellis, S. Mazzocchi, R. Polillo – *Schemi di dialogo: uno strumento per la rappresentazione dei dialoghi di un sistema interattivo* – Note di Software n° 4-1977

- CAST 79 – G. Castelli, G. Haus, M. Maiocchi – *Una metodologia di stesura e verifica di specifiche funzionali di procedure e programmi software* – Note di Software n° 10/11 – Aprile/Settembre 1979

- CERI 81 – M. Ceriani, A. Cicu, M. Maiocchi – *A methodology for accurate software test specification and auditing – dal volume "Computer Program Testing"* – B. Chandrasekaran, S. Radicchi (eds.) – North Holland Publishing Company (C) SOGESTA 1981

- CHIA 80 – A. Chiapparino, A. Cicu. M. Maiocchi – *A documentation method and the related tool for a software engineering environment* – vol. II Atti del Convegno A.I.C.A. 1980 – Bologna, Ottobre 1980

- CHOW 76 – T.S. Chow – *A generalized assertion language* – 2nd International Conference on Software Engineering – S. Francisco, Ottobre 1976 – IEEE cat. 76 CH 1125 – 4C

- CICU 70 – A. Cicu – *Il controllo di qualità del software* – Rivista di Informatica (AICA) – vol. 1, n.2 – Ottobre 1970

- CICU 81/1 – A. Cicu. M. Maiocchi – *A method for anatysis, development and quality control of software – Proceedings of Convencion Informatica Llatina* – Barcelona, 1981

- CICU 81/2 – A. Cicu – *The quality of a computer program: the user view and the software engineering view* – dal volume "Computer Program Testing" – editors: Chandrasekaran, Radicchi; Publisher: North Holland, SOGESTA 1981

- COOP 81 – J. Cooper – *Development of MIL-STD-1679 – in Proceedings di (SWEN 81)*

- COUG 73 – J.D. Couger – *Evolution of Business System Analysis Techniques* – ACM Computing Surveys – vol. 5, n° 3 – Sept. 73

- DEGL 76 – G. Degli Antoni, G. Gobbi – *Aspetti relativi alla documentazione dei programmi* – da giornate di studio su "Programmazione strutturata: esperienze e orientamenti" – HISI, Milano, Giugno 1976 – Rivista di Informatica (AICA) – vol. III, n° 4 – Ottobre/Dicembre 1977

- DEGL 77/1 – G. Degli Antoni, G. Torriani – *Documentazione e produttività* – Note di Software n° 3 – Luglio/Settembre 1977

- DEGL 77/2 – G. Degli Antoni, B. Zonta – *Metafora e documentazione* – Note di Sofware n° 4 – 1977

- DEGL 78 – G. Degli Antoni, M. Maiocchi, R. Polillo – *PSPN: un approccio ai problemi di comunicazione dell'informazione tecnica* – Note di Software n. 6/7 – 1978

- DIJK 72 – E.W. Dijkstra – *The Humble Programmer* – Communication of ACM vol. II n. 10 – Ottobre 1972

- DOD 80 – Department of Defence (U.S.A) – *Requirements for Ada Programming Support Environments ("Stoneman")* – Febb. 1980

- DONA 78 – H. Donaldson – *A guide to the successful management of computer projects* – Associated Business Press – London, 1978

- FARN 76 – S. Farnedi, C. Poggiali, R. Polillo, A. Vignoni – *Test Reporting Package (TRP): un sistema per la gestione dei risultati delle prove di un sistema operativo* – Proceedings AICA Symposium – Milano, 1976.

- FIPS – A Nationall Bureau of Standards – *Federal Information Processing Standards* (vedi lista dei FIPS Pubs in Asterisk – SigDoc vol. 7 n. 5 – Novembre 81)

- FIPS 76 – U.S. National Boureau of Standards – *Guideline for Documentation of Computer Programs and automated data Systems* – Code: FIPS PUB 38 - Febbraio 1976.

- FIPS 79 - U.S. National Boureau of Standards – *Guideline for Documentation of Computer Programs and automated data Systems for the Initiation Phase* - Code: FIPS - PUB 64 – Agosto 1979

- FOST 81 – G.N. Foster – *Principles of Software Standardization* – in Proceedings di (SWEN 81)

- GIMP 80 – J.F. Gimpel – *Countour: A method for preparing structured flow-charts* – Sigplan Notices, Vol. 15, n° 10 – Ottobre 1980.

- GLAS 81 – R.L. Glass – *Standards for standards writers* – in Proceedings di (SWEN 81)

- GOBB 77/1 G. Gobbi, L. Trabattoni – *Macro operazioni* – Note di Software n° 2 – Aprile 1977

- GOBB 77/2 G. Gobbi, M. Maiocchi – *Autodocumentazione dei programmi* – Note di Software n. 3 – Luglio/Settembre 1977

- GOBB 78 – G. Gobbi, M. Maiocchi – *A guide to self-documented programs* – Asterisk, System Documentation Newsletter ACM/SigDoc – vol. 5 n. 4 – Novembre 1978

- GOOS 75 – G. Goos – Documentation – chapter 4 B of "Software Engineering – An advanced course – (Lecture Notes in Computer Science – vol. 30 – Springer Verlag – 1975

- HIEM 75 – P. Hiemann – *A new look at the Program Development Process – "Programming metholodogy"* – vol. 23 of Lecture Notes in Computer Sciences – Springer Verlag – 1975

- HISI 79 – Manuale di Ingegneria di Software HISI vol. 1: il ciclo di produzione – vol. II: gli standards di documentazione – codici 3903079P, 390380V – Laboratorio di Ingegneria di Software della Honeywell Information Systems Italia – Pregnana Milanese

- HOBB 77 – J.R. Hobbs – *What the nature of natural language tells us about how to make natural – language – like programming languages more natural* – SIGPLAN (ACM) vol. 12, n° 8 - 85-93

- IEEE 81/1 – Standards P730 – *Standard for Software Quality Assurance Plans* – ottenibile da IEEE dard Board – Software Engineering Standard Subcommittee – 58 Lambert Johnson Drive, Ocean, NJ 07712

- IEEE 81/2 – *Software Engineering Standards Application Workshop* – August 18/20 1981 – S. Francisco, Ca – organizzato da IEEE Computer Society – IEEE cat. n° 81 CH 1633-7

- IEEE 81/3 – C. Jones (editor) – *Tutorial on Programming Production: Issues for the Eighties* – IEEE Computer Society, IEEE Catalog EHO 186-7, 1981

- INGR 78 – F.S. Ingrassia – *Combating the "90% complete" Syndrome* – Datamation – vol. 24, n° 1 – Gennaio 1978 – 171-176.

- JACK 75 – M.A. Jackson – *Principles of program design* – Academic Press – London, New York, 1975

- JONE 81-C. Jones – *Programming Requirements and Design Methods* – p. 221 in IEEE 81/3

- KATZ 71 – J. Katzenelson – *Documentation and the Management of a Software project – A case study* – Software Practice and Experience – vol. 1, 147-157 - 1971

- KATZ 76 – H. Katzan Jr – *System design and documentation. An introduction to the HIPO Method* – Van Nostrand Reinhold Company – 1976

- KURI 81 – T. Kurihara, S.T. Redurine, S.A. Zaveler – *Observations on documentation standards revision: FIPS PUB 38 after four years* – in Proceeding di (SWEN 81)

- LANO 79 – R.J. Lano – *A technique for software and systems design* – TRW Saigon Software Tecnology – vol. 3 – (Ed. B.W. Boehm) – North Holland 1979

- LANZ 77 – G.A. Lanzarone, M. Maiocchi, R. Polillo – *Introduzione alla Programmazione Strutturata* – Collana dei Quaderni di Informatica – ed. Franco Angeli – 1977

- LECH 67/1 – C.P. Lecht – *The management of Computer Programming Projects* – American Management Association – 1967

- LECH 67/2 – C.P. Lecht – *Comment Gérer les projects des programmes ordinateur* – Entreprise Moderne d'Edition Paris 1969

- LECL 82 – Y. Leclerc, S.W. Zucker, D. Leclerc – *A Browsing approach to Documentation* – Computer, June 1982 – p. 46-49

- MAIO 79 – M. Maiocchi, E. Spoletini – *PHOS: una evoluzione delle metodologie di programmazione* – Quaderni di Informatica n. 11 – 1979

- MART 78 – K.A. Martin – *Software acceptance testing that goes beyond the look* – Proceedings of the Software Quality and Assurance Workshop – ACM SEN – vol. 3 n° 5 – Novembre 1978

- MASE 77 – A. Maserati, S. Mazzocchi, R. Polillo, G. Torriani – *Documentazione per l'utente: alcune indicazioni e un esempio* – Note di Software n° 3 – Luglio/Settembre 1977

- MATU 76 – D. Matusek – The case for the assert statement – Sigplan Notices, Agosto 1976

- METO 21 – J. Barone, G. Damaggio, P.R. Torrigiani – *Note sulla definizione del ciclo di vita del software* – Doc. METOD 21 – obiettivo METOD, Progetto Finalizzato per l'Informatica – P1 – CNR

- METO 28 – A. Cicu – *Il processo di Produzione Software della HISI* – Doc. METOD 28 – obiettivo METOD – Progetto Finalizzato per l'Informatica – P1 – CNR (è il documento HISI 79 reso disponibile pubblicamente)

- METO 31 A. Cicu, M. Maiocchi – *Una proposta di standards per la documentazione tecnica del software* – Doc. METOD 31 – obiettivo METOD – Progetto Finalizzato per l'Informatica – P1 – CNR – 1981

- METZ 73 – P.W. Metzger – *Managing a programming project* – Prentice Hall Inc. – 1973

- MILL 70 – H. Mills – *Syntax directed documentation for PL360* – Comm. of ACM – 13,4 Aprile 1970

- MILL 75/1 – H.D. Mills, R.C. Linger – *On the development of Systems of Men and Machines* – "Programming Methodology" – vol. 23 of Lecture Notes in Computer Sciences – Springer Verlag – 1975

- MILL 75/2 – H. Mills, R.C. Linger – *Definitional text in structured programming* – COMPCONF 1975

- MILL 80 – H.D. Mills – *Principles of Software Engineering* – IBM Systems Journal – vol. 19,4,1980

- MSTD 74 – Department of Defence, USA – *Military specification* – Software Quality Assurance Program Requirements – MIL-S-52779 (AD) – April 1974

- MSTD 78 – Department of Defense – *Military Standard Weapon System Software Development* – MIL-STD-1679 (Navy), Dec. 1978

- MSTD 79 – Naval Training Equipment Center, Military Standard Trainer System Software Development – MIL-STD-1644 (TD) – March 1979

- NASS 73 – I. Nassi, B. Sneiderman – *Flowchart Techniques for Structured Programming* – ACM SIGPLAN Notices 8, n. 8 – 12-26 Agosto 1973

- NATO 68 – *Nato Science Committee* – Sofware Engineering – Report on Conference in Garmish (Germany) – 7/11 Ottobre 1968 – Editors: P. Naur, B. Randell

- NAUR 74 – P. Naur – *Concise Survey of Computer Methods* – Student litteratur – Lund (Sweden) – 1974

- NDSW 10/11 – Tutto il numero 10/11 – 1979 di Note di Software è dedicato all'impiego di Reti di Petri nella descrizione di programmi e sistemi

- NEWM 76 – N. Newman, T. Lang – *Documentation for Computer Users* – SOFTWARE Pratice and Experience, vol. 6, p. 321-326, 1976

- NEWM 82 – P.S. Newman – *Towards an integrated development environment* – IBM Systems Journal – vol. 21, n. 1 – 1982

- ORR 77 – K.T. Orr – *Structured Svstems development* – Yourdon, Inc. – New York – 1977

- ORR 81 – KO FUTURE (S) – *The Journal of the Management of change* – Summer 1981

- PARN 72/1 – D.L. Parnas – *A Technique for Software Module Specification with examples* – in Comm. of ACM – May 1972 – vol. 15, n. 5

- PARN 72/2 – D.L. Parnas – *On the criteria to be used in decomposing systems into modules* – in Comm. of ACM – Dec. 1972 – vol. 15 n. 12

- PETE 81 – L. Peters – *A conceptual basig for software design standards* – in Proceedings di (SWEN 81)

- ROSS 76 D.T. Ross – *Homilies for Humble Standards* – Comm. of ACM – vol. 19, n° 11, Novembre 1976

- ROSS 78 D.T. Ross – *Structured Analysis (SA): a language for communicating ideas* – from Gries – "Programming methodology texts and monographing" Computer Science-Springer Verlag – 1978

• SAMM 66 – J.E. Sammet – *The use of English as a Programming Language* – Comm. of ACM – vol. 9, n° 3 – March 1966

• SEGA 81 – A.R. Segal (ed.) – *Proposal: evalutation of environments for the Software Engineering Process* – in (SIGS 81) qui citato

• SHEP 81 – S.B. Sheppard, E. Krussi, B. Curtis – *The effects of symbology and spatial arrangement on the comprehension of software specifications* – Proceedings of the Fifth International Conference on Software Engineering, 1981, pp. 207-214 (reprinted in IEEE 81/3)

• SHNE 77 – B. Shneiderman, R. Mayer, D. McKay, P. Heller – *Experimental Investigations of the Utility of Detailed Flowcharts in Programming* – Commit. of ACM – June 1977, vol. 20, n° 6

• SIGS 81 – SIGSOFT (Special Interest Group – ACM) – *First Software Engineering Symposium on Tool and Methodology Evalutation* – Pingree Park, Colorado – 9/11 Giugno 1981 – Software Engineering Notes – vol. 7, n° 1 – Gennaio 1982

• SOFT 76 – Softech Inc. – *An introduction to SADT* – Softech Document – n. 9022-78R – Novembre 1976

• SOUT 78 – R.N. Southworth – Responding to MIL-S-52779 – *Proceedings of Software Quality Assurance Wokshop* – Novembre 15/17 1978 – ACM – San Diego (SEN vol. 3, n° 5)

• STAY 76 – J.F. Stay – *HIPO and Integrated program design* – IBM Systems Journal 15 n° 2 – 143-154 - 1976

• STUC 75 – L.C. Stucky, G.L. Foshee – *New assertion concepts for self-metric validation* – International Conference on Reliable Software – Los Angeles – Aprile 1975

• SUKE 81 – A.N. Sukert, A.L. Goel – *A MIL-HDBK for software reliability assessment: progress and perspectives* – in Proceedings di (SWEN 81)

• SWEN 81 – *Software Engineering Standard Application Workshop* – 18-20 Agosto – S. Francisco – IEEE 81 CH1633-7 – 1981

• TAUS 77 – TAUS 79 – R.C Tausworthe – *Standardized Development of Computer Software* (Part 1: Methods 1977 – Part 2: Standards 1979) – Prentice Hall

• TEXT 81 – Proceeding of the ACM SIGPLAN SIGOA Symposium on text manipulation – Portland, Oregon – 8/10 Giugno 1981 – SIGPLAN Notices ACM – vol. 16, n. 6 – Giugno 1981

• TEXT 83 – *Atti del Convegno AICA su "Text processing"* – Milano – Palazzo ex-Stelline – Novembre 1983

• TRAU 73 – H. Trauboth – *Guidelines for documentation of Scientific Software Systems* – 1973 IEEE Symposium on Computer Software Reliability – Aprile 30 / Maggio 2, 1973 – New York – IEEE cat. 73 CH 0741 – 9CSR

• TRIP 81/1 – L.L. Tripp (ed). – *Evalutation of Software Development cycle methodology implementation* – in (SIGS 81) qui citato

• TRIP 81/2 – L.L. Tripp – *Top down, bottom up approach to software engineering standards* – in Proceeding (SWEN 81)

• YODE 78 – C.M. Yoder, M.L. Schrag – *Nassi-Shneiderman Charts: An Alternative to Flowcharts for Design* – ACM Proceedings SIGSOFT/SIGMETRICS Software and Assurance Workshop, Nov 1978 (Reprinted in IEEE 81/3)

• YOHE 74 – J.M. Yohe – An overwiew of Programming Practices – ACM Computing Surveys – vol. 6, n° 4 – Dicembre 1974

• WALS 69 – D.A. Walsh – *A guide for Software Documentation* – New York 1969

• WEDB 81 – G.H. Wedberg – Implementation of software engineering standards – in Proceedings (SWEN 81)

STRUMENTI GRAFICI DI SOFTWARE DESIGN

F. Capozza, C. Gallo, F. Esposito
Istituto di Scienze dell'Informazione, Università di Bari

0. INTRODUZIONE

Le rappresentazioni grafiche o "disegni" rivestono una rilevanza consolidata in tutti i rami dell'ingegneria, per l'intriseca efficacia della forma espressiva (un grafico si sviluppa su due dimensioni, mentre i testi alfanumerici solo su una) che consente, tra l'altro, rapidità di lettura ed ampiezza di visione.

In linea di principio nell'ingegneria del software le rappresentazioni grafiche sono ancora più importanti per una serie di motivi, tra i quali:

- il disegno del software può essere ritenuto il prodotto principale, poichè si può ipotizzare la produzione automatica del codice a partire da un disegno formale e completo;
- il disegno costituisce la parte essenziale della documentazione, che a sua volta può essere ritenuta il prodotto software nel suo complesso (in un'ottica particolare si possono ritenere coincidenti il software, il suo disegno e la sua documentazione);
- la produzione del software è costituita essenzialmente da fasi di analisi e disegno;
- l'analisi e il disegno del software implicano un complesso di comunicazioni fra l'utente (o committente) e/o gli addetti ai lavori (analisti e/o programmatori), con intriseche difficoltà connesse alle differenze di punto di vista, di cultura e di visione dei soggetti interessati.

Ci si può porre il quesito se esiste e sia possibile definire un insieme di strumenti grafici che soddisfi le esigenze di analisi e disegno software, ed in particolare:

- sia completo, cioè in grado di rappresentare i vari aspetti (flusso di controllo, flusso di dati, concorrenza, etc.) per i differenti tipi di software;
- sia quanto più possibile formale, in modo da non lasciare spazio ad ambiguità di interpretazione.

Nel tentativo di rispondere a tale quesito si è condotto uno studio articolato nei seguenti passi:

- classificazione preliminare delle esigenze di rappresentazione nelle fasi di analisi e disegno con riferimento ai diversi tipi di software (EDP, real-time, etc.);
- valutazione dell'efficacia di rappresentazione delle differenti esigenze da parte degli strumenti grafici di disegno sofware più diffusi o inseriti nelle metodologie di disegno. (*)

1. ESIGENZE DI DISEGNO NEL SOFTWARE

Basandoci sulle scelte del Progetto Finalizzato Informatica, distinguiamo i seguenti tipi di software:

a) software EDP (o gestionale, o commerciale);
b) » real-time;
c) » telefonico;
d) » diagnostico (dell'hardware);
e) » di sistema (sistemi operativi).

Nella realtà produttiva attuale ciascun tipo di software è caratterizzato da aspetti ed esigenze peculiari, ma non esclusivi, i quali hanno indirizzato ad approcci e metodologie di disegno differenti.

Non appaiono invece significative, ai fini della individuazione di strumenti "ingegneristici" di disegno, le dimensioni dei sistemi software, le quali si ripercuotono essenzialmente sui problemi di project-management e sulla complessità dell'organizzazione, sul numero di relazioni ed entità delle comunicazioni e dei flussi informativi.

Naturalmente l'utilità degli strumenti di rappresentazione formale del disegno aumenta con la complessità del software e della "software-factory" (ad es. co-

(*) Si precisa che le valutazioni si riferiscono agli strumenti grafici e NON alle metodologie nel loro coplesso.

municazioni di modeste dimensioni, sono pericolose e da evitare con il crescere della complessità del sistema).

Disegnare, in senso lato, può comprendere ogni attività di "modellizzazione" di entità e relazioni tra esse. Tutte le fasi del ciclo di sviluppo del software, dal riconoscimento dei requisiti dell'utente alla codifica vera e propria dei programmi, richiedono attività di "modellizzazione" e quindi di "disegno".

Nell'ambito degli strumenti di modellizzazione, secondo una tradizione ingegneristica consolidata, quelli "grafici" hanno una particolare rilevanza pratica.

1.1 Fasi del ciclo di vita del software

Il ciclo di vita del software si può scomporre nelle seguenti fasi fondamentali:

- ANALISI E DEFINIZIONE DEI REQUISITI
  L'utente di un sistema avverte la necessità di passare ad una gestione automatica e quindi pone delle richieste, la cui natura va analizzata; la struttura principale del tipo di sistema che soddisfa (o dovrebbe soddisfare) i suoi bisogni viene dunque schematizzata, e viene sviluppata la descrizione funzionale del sistema, insieme ai vincoli sulla sua struttura e sull'uso delle risorse.

- DISEGNO DI STRUTTURA
  A partire dai requisiti, viene sviluppato un disegno globale del sistema che soddisfi la struttura del problema. A questo livello vengono evidenziate le componenti (funzioni) del sistema e le loro interrelazioni (intefacce), il flusso di controllo, la descrizione strutturale ed il flusso dei dati.

- DISEGNO ESECUTIVO
  Le parti principali del disegno vengono maggiormente dettagliate, insieme alla precisa definizione degli algoritmi, delle strutture dei dati e delle interfacce tra i vari componenti individuati. Il disegno dettagliato può richiedere parecchi livelli di raffinamento, in ciascuno dei quali (come pure in tutte le altre fasi) il disegno prodotto ad un livello deve essere chiaro e comprensibile (come un progetto ingegneristico – da cui il concetto di "software-blueprint") al livello successivo.
  Ciascun livello di raffinamento deve operare un feed-back rispetto al livello precedente, in cui il responsabile deve verificare la corretta interpretazione e realizzazione di quanto progettato e comunicato al livello inferiore. Questo modo di operare, oltre ad indurre ad una maggiore partecipazione tra i vari gruppi di lavoro, consente di rilevare al più presto possibile il maggior numero di errori progettuali (il cui costo di eliminazione cresce con il ritardo di rilevamento).

Successive al disegno sono le fasi finali del ciclo di sviluppo del software.

- REALIZZAZIONE (IMPLEMENTATION)
  Produce la realizzazione fisica del disegno, comprendente la codifica, il testing dei singoli componenti e la loro integrazione, l'introduzione di eventuali opportune modifiche, il testing del sistema completo e la valutazione delle prestazioni.

- MANUTENZIONE
  Riguarda qualunque intervento sul software successivo all'installazione del sistema (individuazione e rimozione di errori, aggiunte di nuove funzioni, modifiche di funzioni esistenti, etc.).

1.2. Esigenze di disegno delle varie fasi

Si possono individuare le seguenti esigenze di descrizione e rappresentazione in relazione alle fasi di sviluppo del software.

1.2.1 Fase di Analisi

Definizione del modello dell'ambiente visto dall'utente: passo iniziale per lo studio e il successivo sviluppo del software. Tale attività perviene alla rappresentazione del:

- modello strutturale, ovvero descrizione dei centri di informazione ("entità") e loro "relazioni";

- modello dinamico, ovvero descrizione dei centri di trattamento dell'informazione ("processi") e "azioni" da questi svolte;

- descrizione e rappresentazione dei vincoli, quali sincronizzazione (*) e vincoli temporali (di prestazione). La corretta individuazione dei vincoli completa la definizione della struttura del problema, ed è essenziale per soddisfare le aspettative dell'utente;

- individuazione e descrizione delle interfacce esterne (in particolare verso gli "utenti"), fase nella quale si pongono gli utenti nella condizione di prendere conoscenza dell'aspetto esterno e delle modalità di utilizzo.

1.2.2 Fase di disegno

La fase di disegno produce un modello di architettura

(*) i vincoli di "sequenza" sono inclusi in quelli di sincronizzazione.

("struttura" e "comportamento") del software a partire dalle indicazioni dell'analisi di fattibilità (che tiene conto del modello dell'ambiente) e dei vincoli individuati nell'attività di analisi.

La definizione del modello implica:

- il disegno delle funzioni;
- la descrizione dei dati;
- la rappresentazione dei vincoli di sincronizzazione e temporali tra i vari processi del sistema ;
- il disegno delle interfacce tra componenti del software.

Il processo di disegno è iterativo ("top-down" o "botton-up" o "misto") perchè ad ogni passo vi è un limite pratico alla complessità che si riesce a gestire.

Il processo di disegno non si esaurisce fin quando vi sono processi decisionali (creativi) in atto.

1.2.3. Riepilogo esigenze.

Le esigenze nel disegno del software possono riepilogarsi nella rappresentazione di:

- entità e relazioni tra entità;
- processi e azioni svolte dai processi;
- interfacce esterne, con particolare riferimento all'interfaccia con l'utente;
- flusso di controllo;
- strutture dati;
- flusso dei dati;
- stati di evoluzione del sistema (stati di processo);
- interfacce tra le componenti del sistema (sottosistemi, moduli, processi, task etc.);
- vincoli di sincronizzazione (e di sequenza);
- vincoli di prestazione (temporali)

Vi sono inoltre diffuse esigenze di comunicazione sia verso l'utente (o committente) che nell'ambito dei gruppi di lavoro.

1.3. Esigenze di disegno in relazione ai tipi di software

1.3.1. Software EDP

È caratterizzato tradizionalmente da elaborazioni di tipo sequenziale, riconducibili a flussi di dati tra centri informativi ("entità"), ed a "transazioni" (trasformazioni che hanno luogo nei centri).

In questo tipo di software l'attenzione va posta essenzialmente sulla struttura dei dati, e quindi obiettivo primario di uno strumento di disegno in quest'area è in genere la definizione delle relazioni strutturali tra i dati e del flusso sequenziale di trasformazione degli stessi.

Metodologie di disegno molto diffuse quali Warnier (LCP [22]) e Jackson (JSP [19]) partono dalle considerazioni precedenti, anche se l'evoluzione del software EDP (ad. es. requisiti di real-time e di concorrenza) ha indotto progressivi ampliamenti delle metodologie per affrontare le esigenze sopravvenute (LCS [23] e JSD [20]).

Nel software EDP non esiste una significativa differenziazione di problematiche fra le varie fasi di analisi e di disegno, essendo ripetitiva l'impostazione in un ambito "input-process-output".

1.3.2 Software real-time

Il software real-time è caratterizzato dalla concorrenza di attività e dall'elaborazione e risposta "in tempo utile" del sistema agli stimoli esterni.

Sorge quindi la necessità di strumenti per esprimere il comportamento dei sistemi che coinvolgono attività concorrenti. Il procedere di attività concorrenti pone problemi di sincronizzazione nell'allocazione delle risorse, nella sequenza di operazioni e nella comunicazione tra processi.

La complessità delle relazioni da considerare rende pressochè indispensabili (oltre che maggiormente utili) strumenti di definizione formale, in particolare nelle fasi di analisi e di disegno nei livelli più alti.

Uno dei concetti principali introdotti ai fini di un'analisi astratta e di una definizione formale dei sistemi real-time è il "processo", quale entità base e funzionalmente autoconsistente. Un processo è caratterizzato dalla funzione, dai dati di I/O e dagli "stati" iniziale e finale.

I meccanismi di interazione tra processi sono essenzialmente due:

"comunicazione": un processo trasmette dati ad un altro;

"sincronizzazione": i processi interagiscono per assicurare una corretta sequenza di eventi per esigenze di "precedenza" o di "mutua esclusione".

1.3.3. Software telefonico

Le esigenze di disegno nel software telefonico sono collegate strettamente ai problemi di strutturazione dei sistemi di comunicazione.

I sistemi di comunicazione sono caratterizzati da scambi di "messaggi" in architetture reticolari.

In tale ambiente sorgono specifiche esigenze di rappresentazione di rete, oltre ai problemi di concorren-

za, sincronizzazione o "cooperazione" analoghi a quelli del software real-time.

1.3.4. Software diagnostico (dell'hardware)

Il disegno del software diagnostico pone problematiche specifiche a livello metodologico (ad es. simulazione guasti, dizionario dei sintomi etc.) ma non ha particolari esigenze di rappresentazione grafica.

1.3.5 Software di sistema (sistemi operativi)

Il software di sistema è caratterizzato da un input esterno di definizione dei requisiti (derivante da obiettivi commerciali, aggiunta di funzionalità a sistemi esistenti, requisiti scaturiti da idee innovative, etc.) che tende a far svanire la fase di analisi, tranne l'attività di formalizzazione dei requisiti.

Da un punto di vista di disegno, peculiarità del software di base sono l'alto contenuto algoritmico, oltre alle esigenze di schedulazione di processi e allocazione di risorse analoghe a quelle dei sistemi real-time.

1.3.6. Riepilogo esigenze/tipo software

Una valutazione delle esigenze peculiari di rappresentazione grafica richieste dai differenti tipi di software è tracciata in tabella A. Dalla tabella appare, ai fini della problematica di rappresentazione grafica, che:

- il software diagnostico non ha rilevanza specifica;
- il software real-time e quello telefonico presentano esigenze del tutto simili;
- il software EDP e quello real-time (telefonico) presentano generalmente esigenze complementari, tranne che per gli aspetti di descrizione delle interfacce sia esterne che interne;
- il software di sistema, tranne che per gli aspetti di descrizione di struttura dati, coinvolge tutte le esigenze di rappresentazione grafica;
- un software EDP al quale siano richieste caratteristiche e/o prestazioni di tipo real-time, presenta esigenze di rappresentazione di tutti gli aspetti.

	ANALISI			ANALISI E DISEGNO							
ESIGENZE / SOFTWARE	ENTITÀ/ RELAZIONI	PROCESSI/ AZIONI	INTERFACCE ESTERNE(UTENT.)	DEFINIZIONE FUNZIONI	FLUSSO DI CONTROLLO	STRUTTURE DATI	FLUSSO DATI	STATI DI PROCESSO	INTERFACCE (MODULI/PROC)	SINCRONIZZAZ.	VINCOLI TEMPORALI
EDP	XXX	X	XX(X)	X	X	XXX	XXX	X(X)	XXX	X(X)	X
REAL TIME	X	XXX	XX(X)	XX	XXX	X(X)	X(X)	XXX	XXX	XXX	XXX
TELEFONICO	X	XXX	X	XX	XXX	X(X)	X(X)	XXX	XXX	XXX	XXX
DIAGNOSTICO	X	–	X	XXX	XX	–	X	–	X	–	–
DI SISTEMA	XX	XX	XXX	XXX	XXX	X(X)	XX(X)	XXX	XXX	XXX	XX

TABELLA A - Esigenze di disegno in funzione del tipo di software

. il numero di X tra 1-3 indica da esigenze scarse ad esigenze elevate
. ( ) indica tendenze previste

2. STRUMENTI DI DISEGNO

Nei paragrafi seguenti si esaminano gli obiettivi, i pregi e le limitazioni delle tecniche grafiche di rappresentazione di disegno software più diffuse.

Gli strumenti grafici di disegno del software sono raggruppabili in due classi:

- diagrammi a blocchi;
- strumenti semigrafici;
- rappresentazioni reticolari.

2.1 Diagrammi a blocchi

Sotto questo termine si raggruppano tutte le varianti del classico diagramma a blocchi o di flusso (flow-chart) che ha per così lungo tempo accompagnato gli sforzi dei programmatori verso la realizzazione di programmi e che ha una vasta applicazione nell'ambiente software.

L'obiettivo inizialie dei flow-chart era rivolto ad una descrizione grafica di algoritmi più leggibile (meno "crittografica") delle sequenze di istruzioni macchina o di linguaggi di programmazione di basso livello (v. figg. 1).

I flow-chart presentano in linea di massima l'inconveniente di non evidenziare la struttura del problema all'aumentare della complessità del disegno. È generalmente difficile comprendere da un flow-chart il funzionamento di un sistema o valutare gli effetti di eventuali modifiche.

I diagrammi a blocchi sono uno strumento efficace per descrizioni sintetiche ad alto livello, ma diventano complessi da disegnare ed interpretare per descrizioni dettagliate.

Le caratteristiche dei diagrammi a blocchi sono riepilogate nei seguenti punti:

- descrizione essenzialmente di algoritmi o di flussi di dati;
- esposizione strettamente sequenziale (manca la possibilità di descrivere il fattore tempo, i vari tipi di vincoli, etc.);
- tendenza a diventare rapidamente complessi e poco leggibili (non sono adeguati a rappresentare con immediatezza composizioni di strutture di controllo di linguaggi ad alto livello);
- scarsa leggibilità e complessità grafica (se in scala 1/1 sono talvolta meno leggibili di un programma scritto in linguaggio ad alto livello);

I diagrammi a blocchi sono stati, in fasi successive, estesi e/o adattati alla rappresentazione di vari aspetti del software.

Tra i molti tipi di diagrammi a blocchi che sono stati proposti si individuano due classi:
. diagrammi a blocchi "generici", utilizzati come strumenti di base e variamente combinati nelle metodologie;
. diagrammi "specifici", che costituiscono elemento caratterizzante di una metodologia.

2.1.1. Diagrammi a blocchi "generici"

Tra i diagrammi a blocchi "generici" si individuano:

- diagrammi di flusso (flow-chart) di controllo (v. fig. 1a);
- flow-chart strutturati (ad es. Warnier flowchart di fig. 1b, HOS flow-chart di fig. 1c e Chapin chart di fig. 1d);
- diagrammi INPUT/PROCESS/OUTPUT, che evidenziano le relazioni di I/O ed il flusso di controllo (ad es. diagrammi funzionali HIPO, v. fig. 1e);
- diagrammi di FLUSSO DATI (data-flow chart o bublle-charts di fig. 1f);
- diagrammi gerarachici di STRUTTURA, i quali evidenziano la suddivisione gerarchica delle funzioni di un sistema (v. fig. 1g);
- diagrammi di FLUSSO-STRUTTURA, i quali descrivono contemporaneamente sia la dipendenza gerarchica dei moduli che il flusso di controllo (v. structure chart di fig. 1h e diagrammi di strutture di fig. 1i);
- diagrammi di INTERAZIONE FUNZIONALE tra moduli, che evidenziano le relazioni e le interfacce tra i moduli (v. anche N2 Charts, par. 2.2.1).

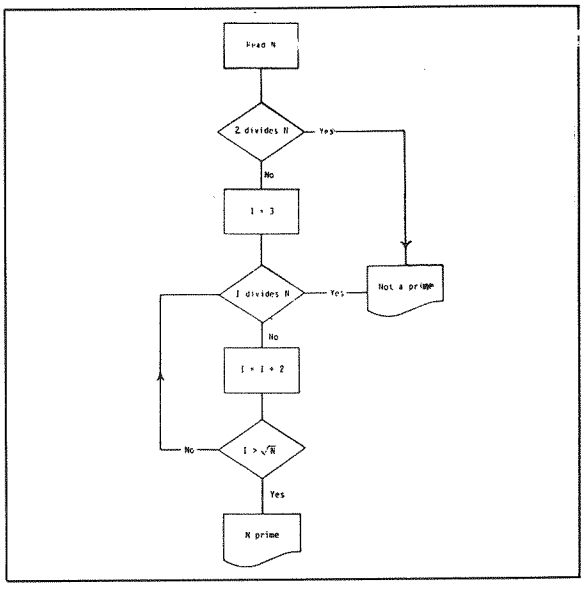


Fig. 1a Flow chart di controllo

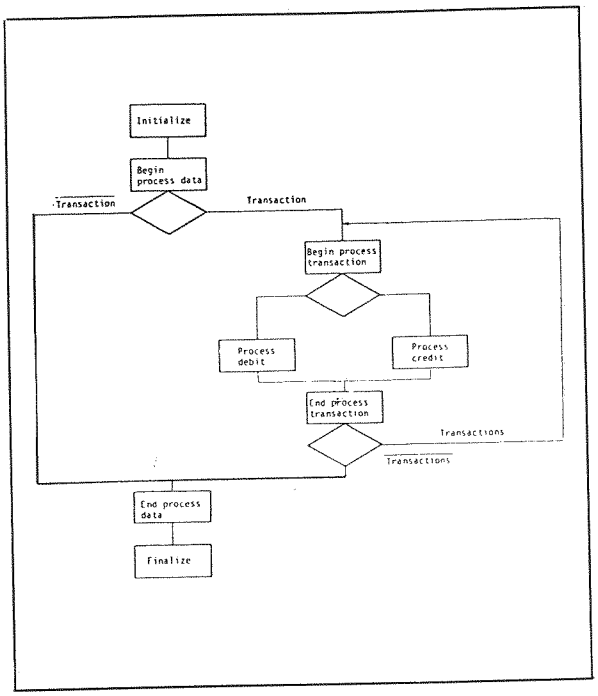


Fig. 1b Diagramma strutturato di Warnier

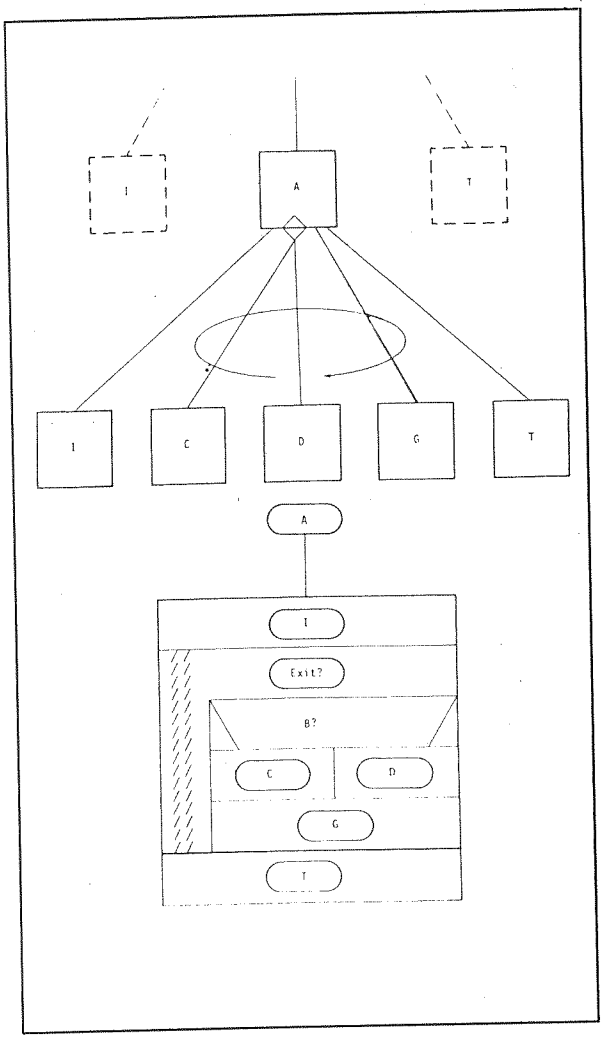


Fig. 1d Chapin chart

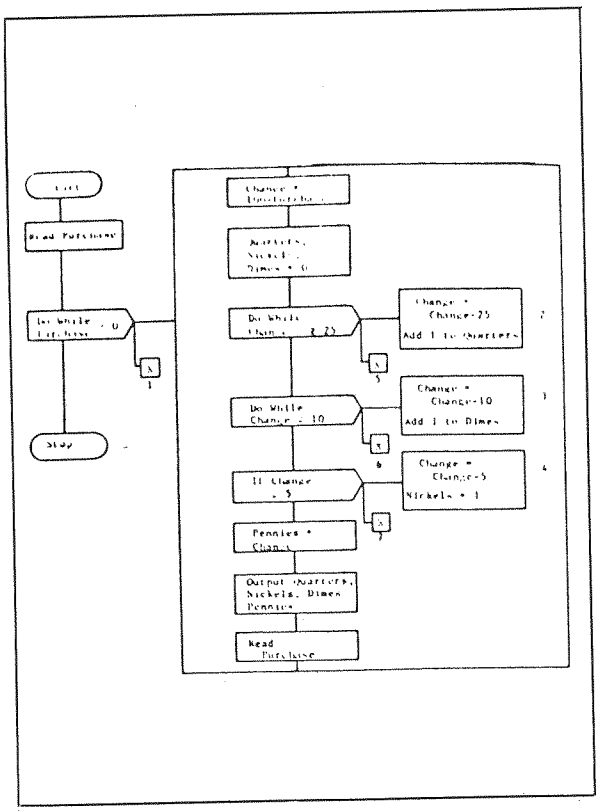


Fig. 1c Hos flow chart

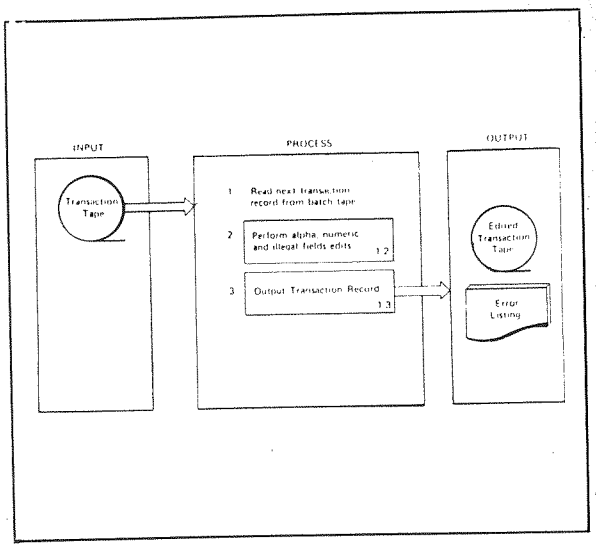


Fig. 1e Diagramma Input/Process/Output

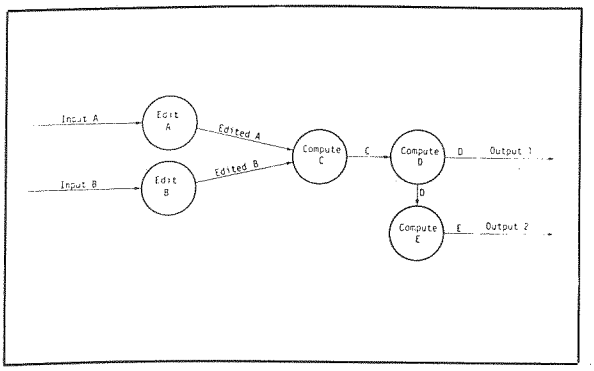


Fig. 1f Bubble Chart

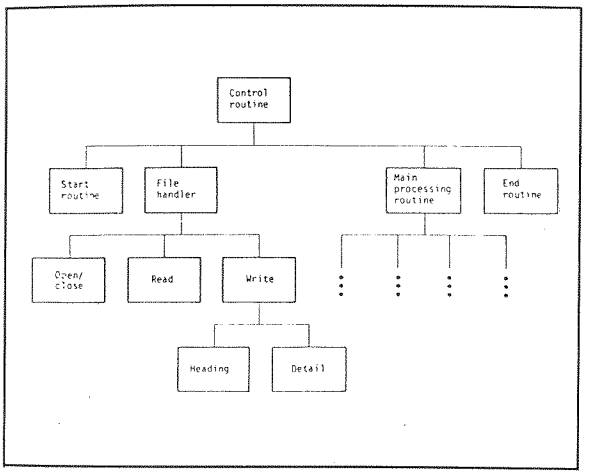


Fig. 1g Diagramma gerarchico di struttura

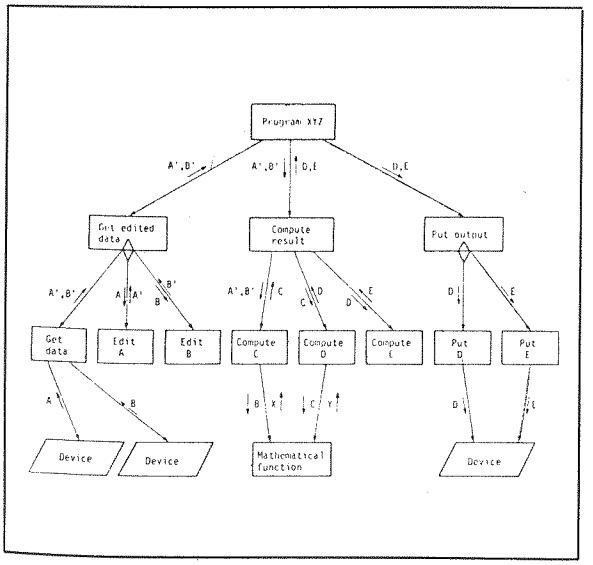


Fig. 1h Structure Chart

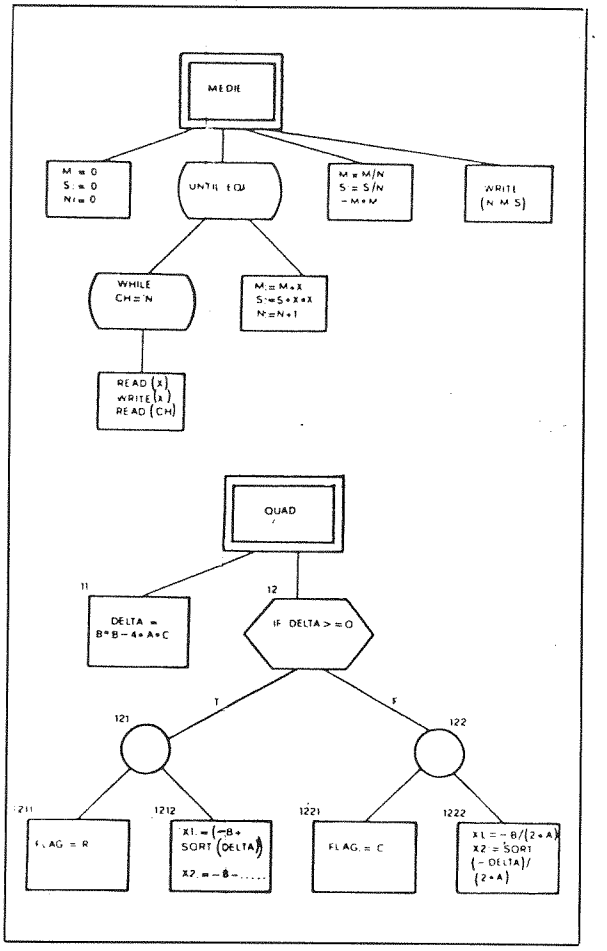


Fig. 1i Diagramma di struttura

2.1.2 Diagrammi a blocchi "specifici"

Tra i diagrammi a blocchi che caratterizzano alcune metodologie di vasta diffusione si individuano:

- diagrammi di JACKSON;
- diagrammi PHOS;
- diagrammi S.A.D.T.;
- diagrammi S.D.L.;
- diagrammi E-R (Entità-Relazioni).

2.1.2.1 Diagrammi di Jackson

I diagrammi di Jackson (JSP) sono strutturati e gerarchici e descrivono (in rappresentazioni complementari) l'organizzazione logica dei dati e il flusso di controllo tramite strutture tipiche della programmazione strutturata (v. fig. 2a, 2b).

Le tre strutture di controllo per la rappresentazione dei dati sono:

- la sequenza di campi di strutture di dati, descritti mediante rettangoli;

- l'iterazione di campi (array o simili) individuata da un asterisco;

- la selezione, rappresentata da un cerchietto in ogni rettangolo alternativo.

L'evoluzione della metodologia di Jackson per il disegno di sistemi (JSD [20]) include ulteriori diagrammi e precisamente:

- diagrammi di modello strutturale, che evidenziano le "entità" e le loro "azioni" (v. fig. 2c);

- diagrammi di modello funzionale, rappresentato come un insieme di processi sequenziali o funzioni ("rettangoli") connessi da code di dati ("cerchi") e/o vettori di stato) ("rombi") (v. fig. 2d). Una particolare entità "clock" e connessioni con vettori di stato "time" introducono eventuali informazioni di temporizzazione.

I diagrammi-dati di Jackson descrivono in maniera semplice le relazoni (statiche) tra i componenti di una struttura dati, ma non definiscono il tipo dei dati.

Pur presentando il vantaggio di una rappresentazione grafica semplice, essi sono meno completi (e quindi meno formali) della definizione di strutture dati tramite linguaggi ad alto livello quali PASCAL, ADA, CHILL, etc.

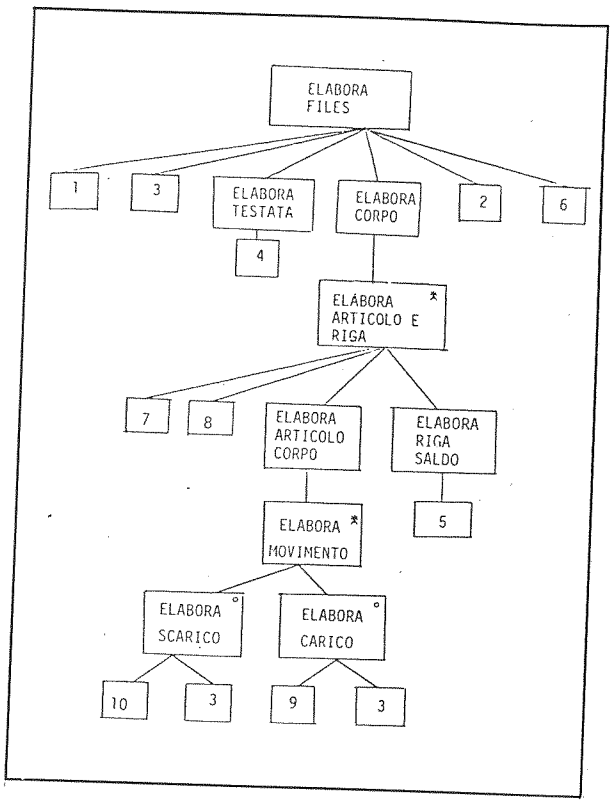


Fig. 2b Diagrammi di struttura di programma di Jackson

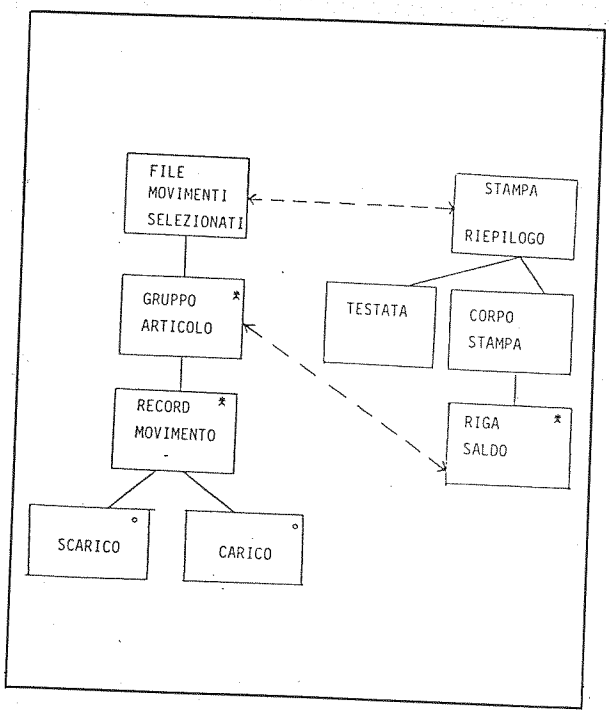


Fig. 2a Diagramma Dati di Jackson

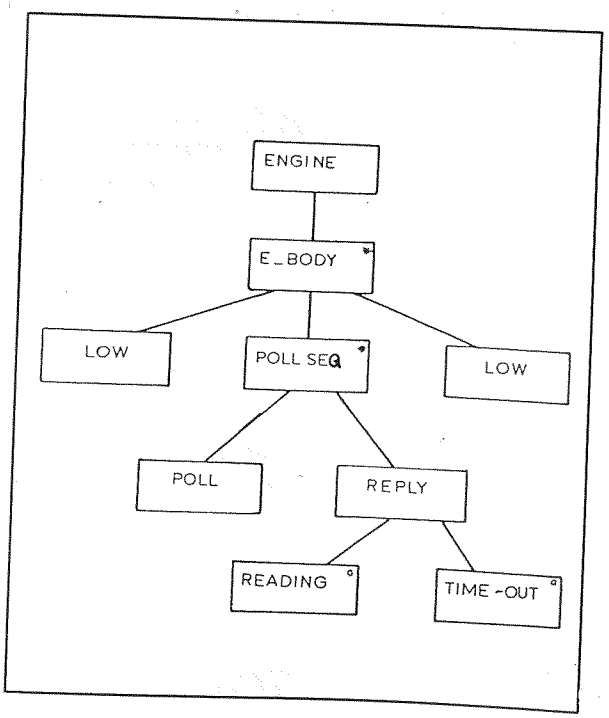


Fig. 2c Diagramma di Jackson di modello strutturale

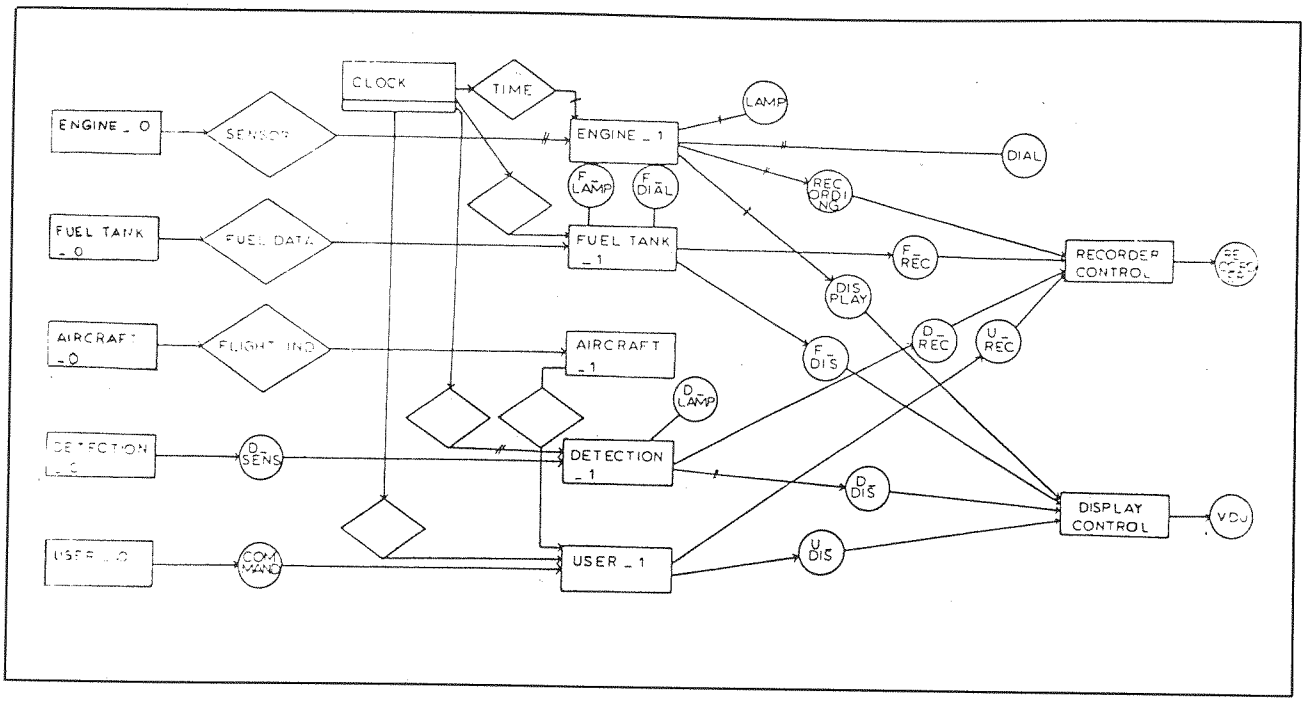


Fig. 2d Diagramma funzionale di Jackson

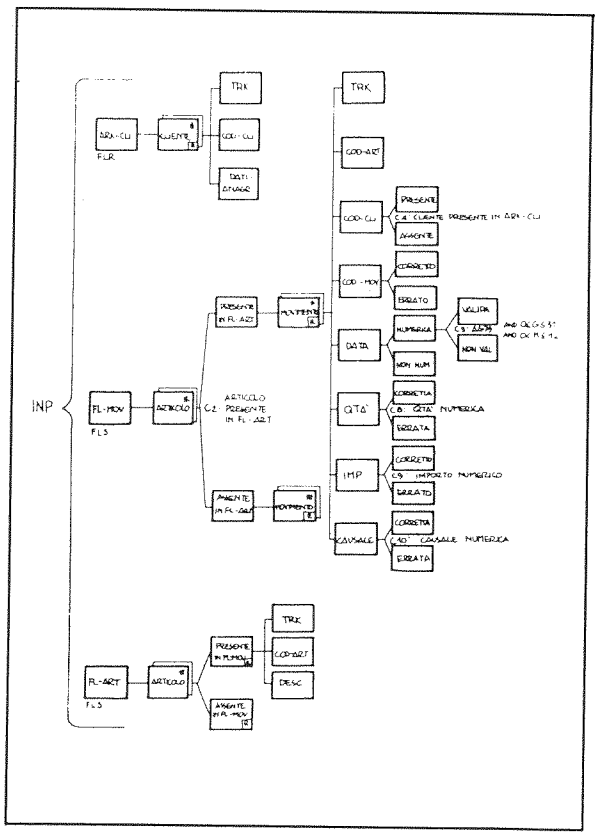


Fig. 3a Diagramma Phos Dati

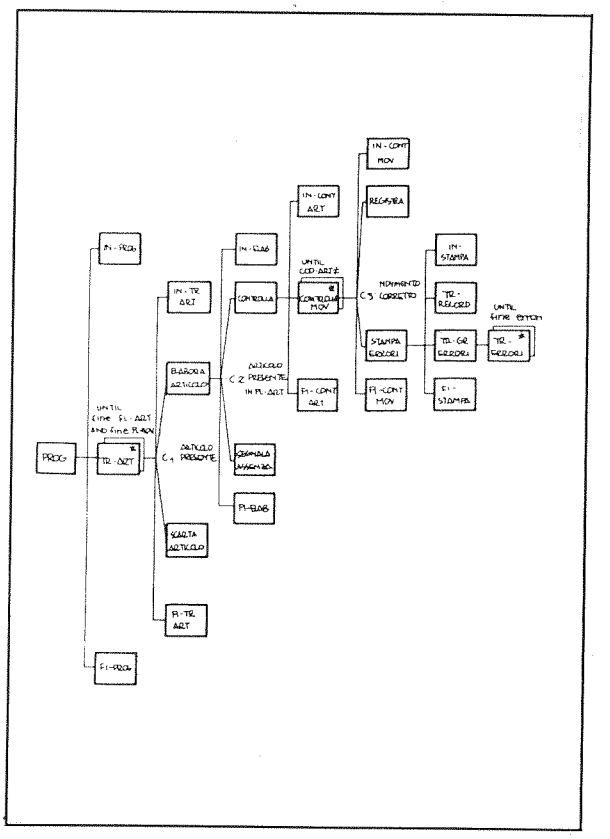


Fig. 3b Diagramma Phos di struttura programma

2.1.2.2. Diagrammi PHOS

I diagrammi PHOS, a parte la differente forma grafica, sono analoghi ai diagrammi dati e di struttura di Jackson (v. fig. 3a,b).

2.1.2.3. Diagrammi S.A.D.T.

I diagrammi S.A.D.T. descrivono il comportamento funzionale dei sistemi (v. figg. 4a,b). Essi sono organizzati gerarchicamente per la progressiva decomposizione di un sistema, e ne descrivono il duplice aspetto attività/dati mediante l'attribuzione di un differente significato ai rettangoli ed alle frecce.

Nei diagrammi "attività" i rettangoli rappresentano le funzioni del sistema e le frecce i dati; queste ultime hanno il significato di input, controllo, output, meccanismo a seconda del lato dal quale entrano/escono. Esse non rappresentano flussi o sequenze, ma interfacce tra le attività (o i dati).

Nei diagrammi "dati" i rettangoli rappresentano i dati (centri di informazione); le frecce individuano le attività che mettono in relazione i dati (attività produttrici dei dati, consumatrici, di controllo e supporto).

La descrizione in due rappresentazioni complementari introduce un certo grado di verifica di congruenza del disegno. Le notazioni utilizzate a complemento della rappresentazione grafica non sono completamente formalizzate.

I diagrammi S.A.D.T. non descrivono la struttura dei dati, nè il flusso di controllo e gli aspetti di concorrenza o sincronizzazione. Essi sono limitati alla descrizone funzionale dal flusso dei dati nella fase di analisi.

2.1.2.4. Diagrammi S.D.L.

I diagrammi S.D.L. (v. fig. 5) sono orientati alla descrizione di processi telefonici. Essi introducono una simbologia specifica per la descrizione degli "stati" dei procesi e dei "segnali" scambiati tra essi. Tale completamento rende i diagrammi S.D.L. idonei alla descrizione del flusso di controllo e di comunicazione e quindi alla rappesentazione degli aspetti di concorrenza e cooperazione tra processi.

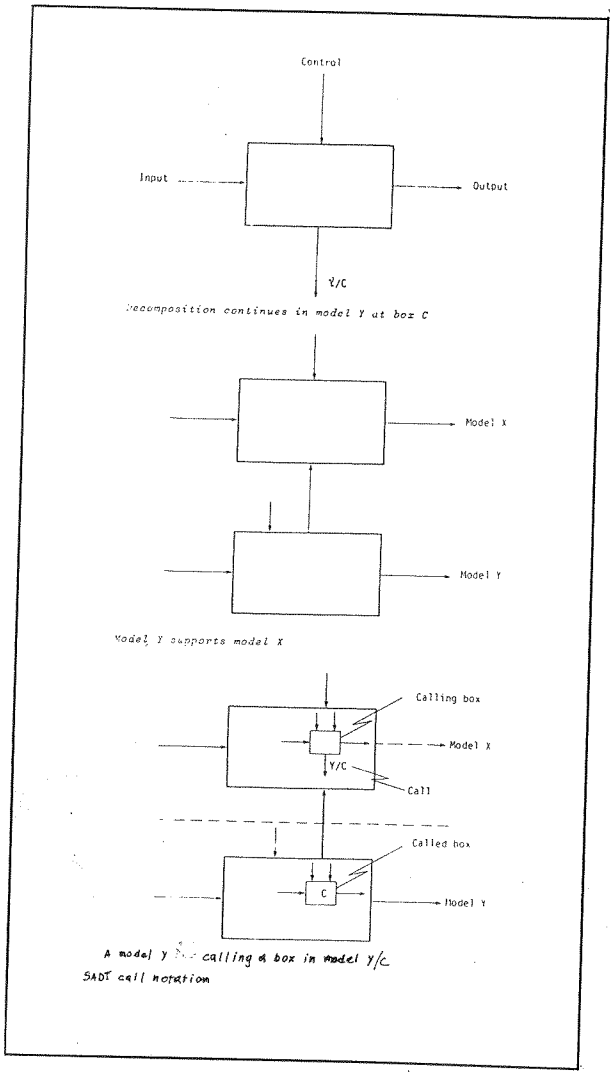


Fig. 4a

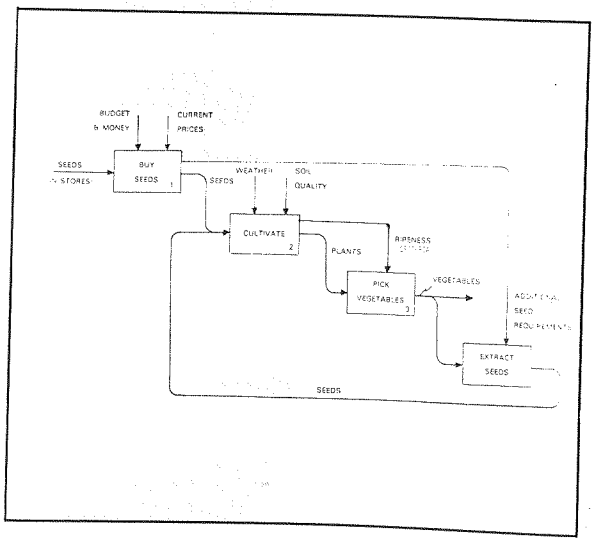


Fig. 4b Diagramma S.A.D.T.

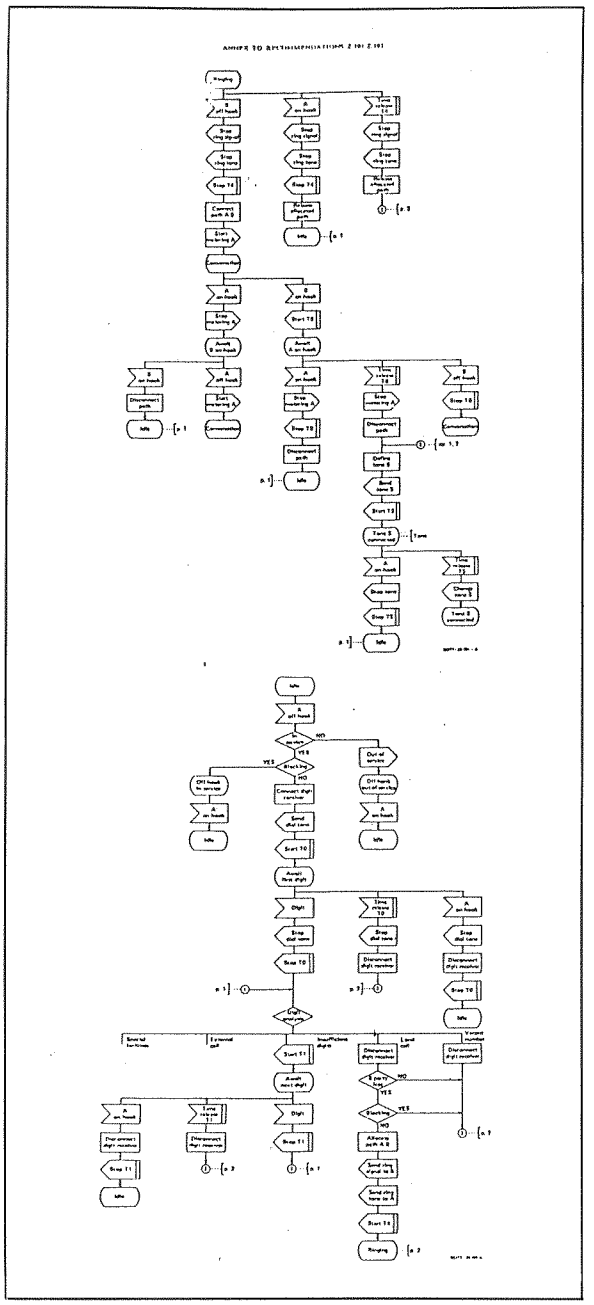


Fig. 5

2.1.2.5. Diagrammi Entità-Relazioni

I diagrammi E-R rappresentano i sistemi come insieme di oggetti elementari ("ENTITÀ") tra i quali intercorrono rapporti di varia natura ("RELAZIONI").

Le entità sono rappresentate da figure rettangolari; le relazioni tra le entità sono essenzialmente rappresentate da figure romboidali, con linee di connessione alle entità messe in relazione.

La presenza di una lettera o numero su una linea indica la cardinalità della relazione (uno-uno, uno-molti, molti-uno, molti-molti). È possibile definire una relazione anche su più di due entità.

Le proprietà delle entità e delle relazioni sono espresse come "attributi" e "valori" delle stesse (ad es. l'entità IMPIEGATO può avere l'attributo ETÀ un cui valore può essere 28). Gli attributi sono rappresentati da frecce dirette dall'entità all'insieme dei valori ammissibili, racchiusi in un cerchio (v. fig. 6a). Le entità e le relazioni vengono identificate mediante un attibuto o combinazione di attributi ("chiave"): ad es. l'attributo MATRICOLA dell'entità IMPIEGATO.

Sono rappresentabili dipendenze tra le entità, quali: la dipendenza di esistenza, (un'entità esiste solo se ne esiste un'altra – rapporto "padre-figlio") e quella di identificazione (un'entità è univocamente riconoscibile solo se rapportata ad un'altra, come per le "vie" nelle "città") (v. fig. 6b).

I diagrammi E-R sono automaticamente traducibili in diagrammi di strutture dati, secondo criteri che tengono conto del tipo di relazioni tra le varie entità.

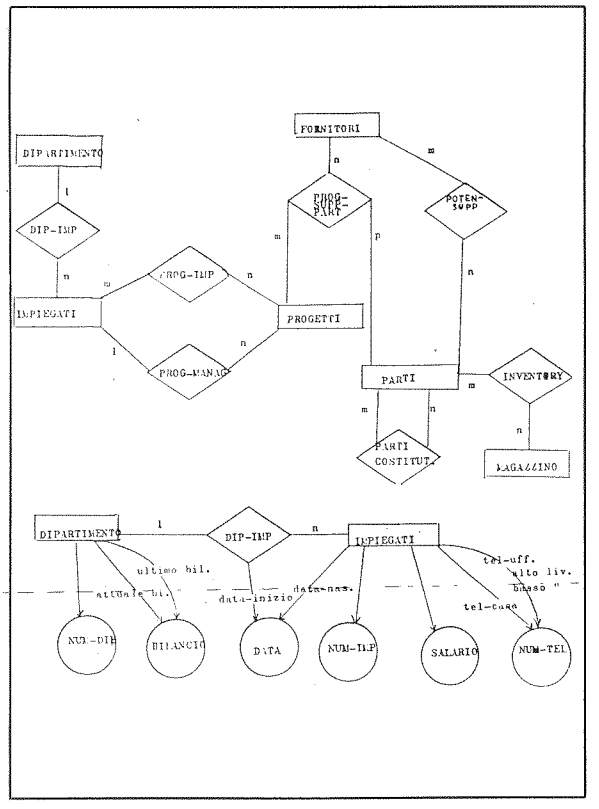


Fig. 6a Diagramma Entità-Relazioni

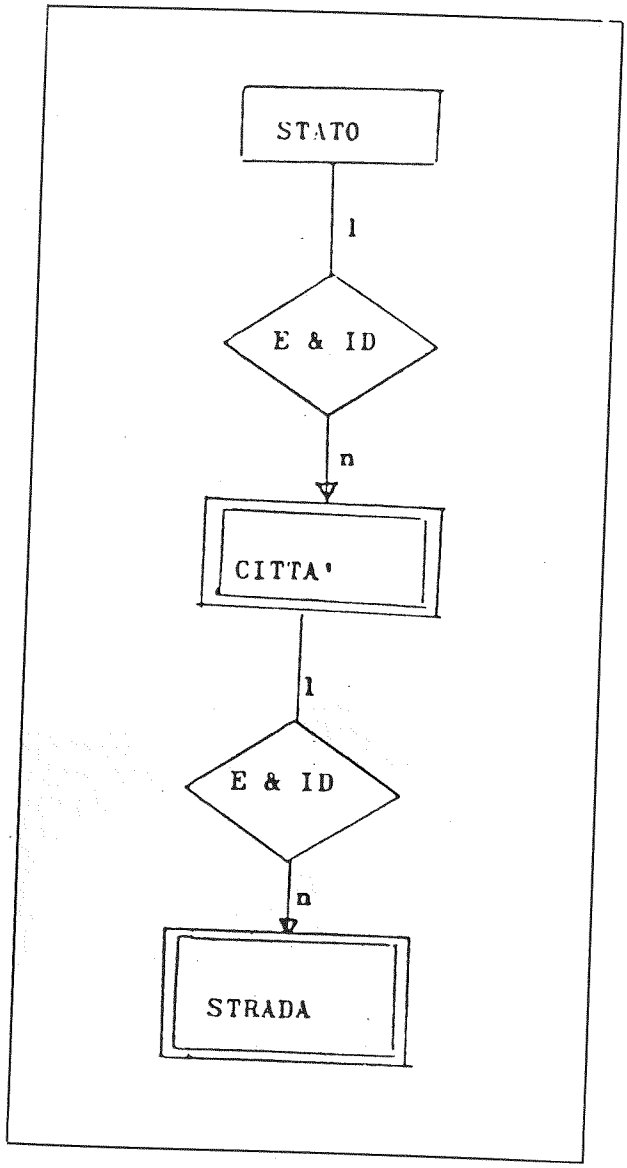


Fig. 6b

2.1.3 Riepilogo dei diagrammi a blocchi

La tabella B riassume le esigenze soddisfatte dai vari tipi di diagrammi a blocchi. In ogni casella la parte superiore si riferisce alla fase di analisi, quella inferiore alla fase di disegno. Con I, S, F si è indicato se lo strumento grafico possiede caratteristiche Informali, Semiformali o Formali.

Nelle ultime due colonne si valuta (con un numero di "x" da uno=scarso a tre=buono) l'efficacia dello strumento grafico quale supporto a comunicazioni esterne (verso il committente) e interne (fra gli addetti ai lavori), limitatamente al contenuto informativo racchiuso in ciascun tipo di diagramma.

Dalla tabella B si rileva una copertura del tutto parziale delle esigenze da parte di ciascun tipo di diagramma a blocchi "generico":

- i diagrammi di Jackson coprono, con diverso grado di accuratezza, quasi tutte le esigenze, con particolare riferimento al software EDP, anche se JSD si estende verso esigenze tipiche di real-time;

- i diagrammi SADT, anche se a livello informale, costituiscono un valido strumento di comunicazione;

- i diagrammi SDL soddisfano gran parte delle esigenze del software telefonico, a livello formale nella maggioranza degli aspetti;

- i diagrammi E-R descrivono in modo formale un aspetto particolare, poco evidenziato dagli altri tipi di diagrammi.

2.2. Strumenti semigrafici vari

Tra gli strumenti di tipo non strettamente grafico (non rientranti, cioè, nella categoria dei diagrammi a blocchi o nella categoria delle rappresentazioni reticolari) si individuano:

- N2 Charts;
- check-list;
- diagrammi di Warnier;
- tavole decisionali.

2.2.1. N2 Charts.

Le N2 Charts sono uno strumento grafico per la tabulazione, definizione, analisi e descrizione di interazioni funzionali ed interfacce.

Esistono vari formati di N2 Charts. Il più semplice è il formato matriciale (v. figg. 7a,b) in cui sulla diagonale principale sono poste le funzioni del sistema, mentre ogni casella della matrice quadrata contiene l'interfaccia (in un senso). Tutti gli output sono definiti sulla riga orizzontale, mentre tutti gli input sono definiti nella colonna. Input ed output esterni al sistema sono definiti nell'area circostante la matrice. Le N2 Charts servono per l'individuazione, oltre che della presenza, della mancanza di interfacce.

Le varianti del formato base sono:

- formato "arrow" (v. fig. 7c), che descrive il flusso e le direzioni delle interfacce;
- formato "step-sequenced" (v. fig. 7d), che descrive le sequenze operative del sistema, ovverossia i vincoli di sequenza;
- formato "loop" (v. fig. 7e), che mostra i principali cicli di iterazione o di feed-back.

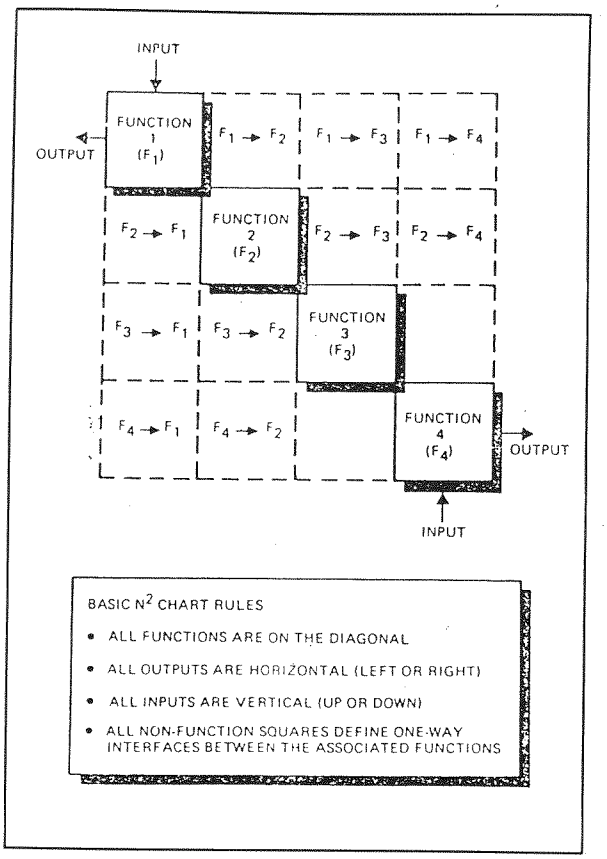


Fig. 7a N^2 Chart

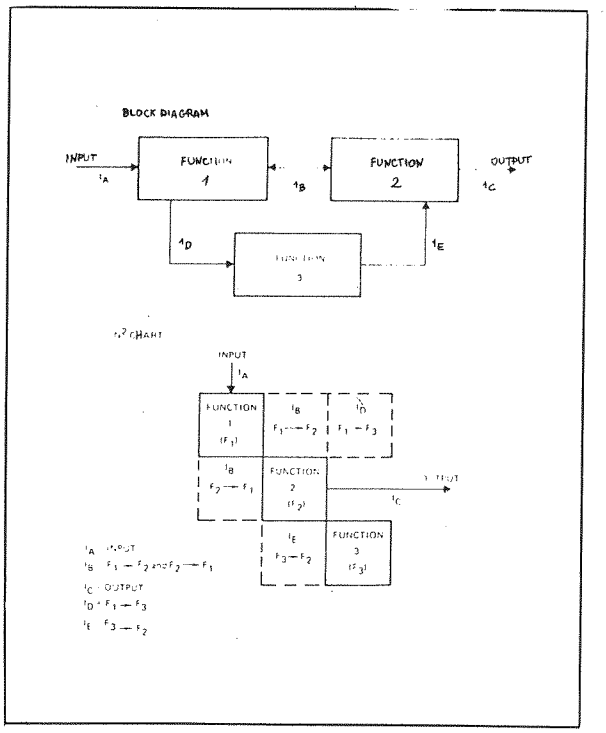


Fig. 7b N^2 Chart formato matriciale

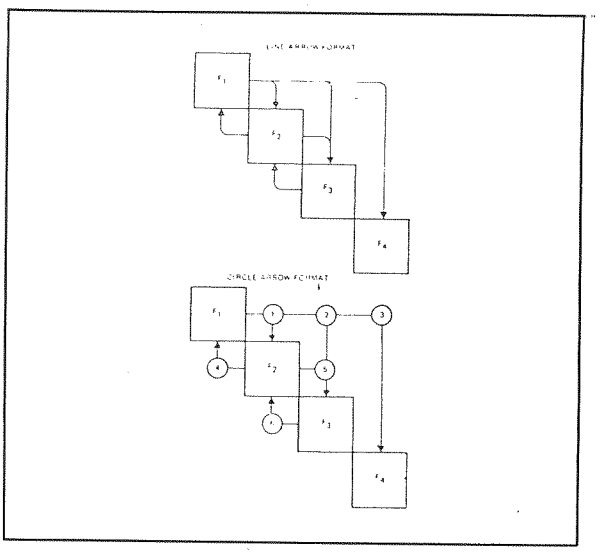


Fig. 7c N^2 Chart formato "ARROW"

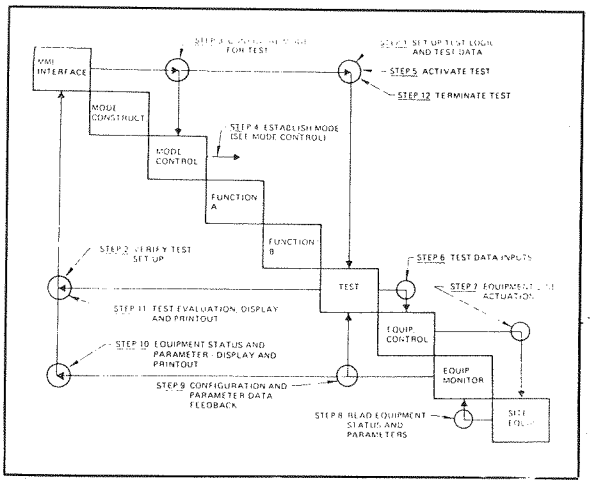


Fig. 7d N^2 Chart formato "Step sequenced"

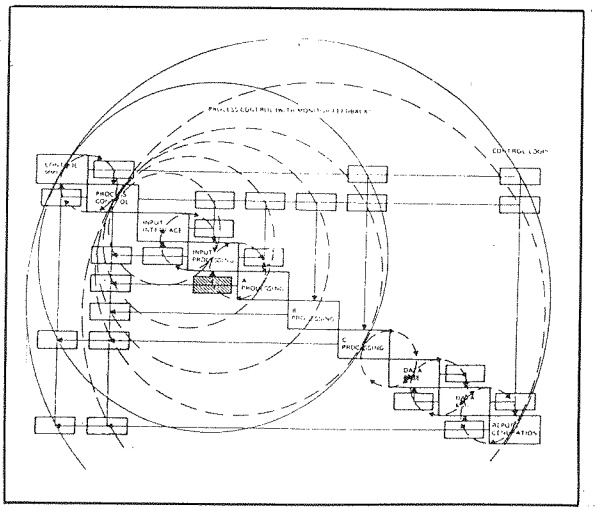


Fig. 7e N^2 Chart formato "LOOP"

ESIGENZE / STRUMENTI	ENTITÀ/ RELAZIONI	PROCESSI/ AZIONI	INTERFACCE EST. (UTENTE)	DEFINIZIONE FUNZIONI	FLUSSO DI CONTROLLO	STRUTTURE DATI	FLUSSO DATI	STATI DI PROCESSO	INTERFACCE (MODULI/PROC)	SINCRONIZZAZZIONE	VINCOLI TEMPORALI	COMUNICAZIONI ESTERNE	COMUNICAZIONI INTERNE	NOTE
FLOW-CHART DI CONTROLLO				I/I	F/F							X	X	
FLOW-CHART STRUTTURATI				I/I	F/F							X	XX	
DIAGRAMMI INPUT/PROCESS/OUTPUT				I/I	I/I		I/I		I/I	I/I		XX	XX	
DATA FLOW-CHART							F/F		I/I			XX	X	
DIAGRAMMI GERARC. DI STRUTTURA				I/I		S/S						X	X	
DIAGRAMMI FLUSSO/STRUTTURA				I/I	S/S		S/S		I/I			XX	XX	
DIAGRAMMI INTERAZ. FUNZIONALE			I	I/I					I/I			X	X	
DIAGRAMMI STRUTT. DI JACKSON	(S)	(S)		I/I	F/F	F/F	I/I	(S)/(S)	I/I	I/I			XX	() SOLO JSD
DIAGRAMMI S.A.D.T		I	I	I/			S/	I/	I/			XXX	XX	
DIAGRAMMI S.D.L.		I		I/I	F/F		S/S	F/F	S/S	F/F		X	XX	ORIENTATO A SEGNALI
DIAGRAMMI E-R	F					F/F						X	XXX	

TABELLA B – Esigenze di disegno soddisfatte dai vari tipi di diagramma a blocchi

2.2.2. Check-list

La Check-list è uno strumento per indicare elenchi di vario tipo, quali ad es. relazioni di Input/Output, interconnessioni tra i vari moduli di un sistema, elenco di errori o situazioni particolari, elenco di dati di Input e/o Output, etc. (v. fig. 8).

Esse sono efficaci come strumento di controllo, purchè la loro forma grafica sia semplice ed espressiva.

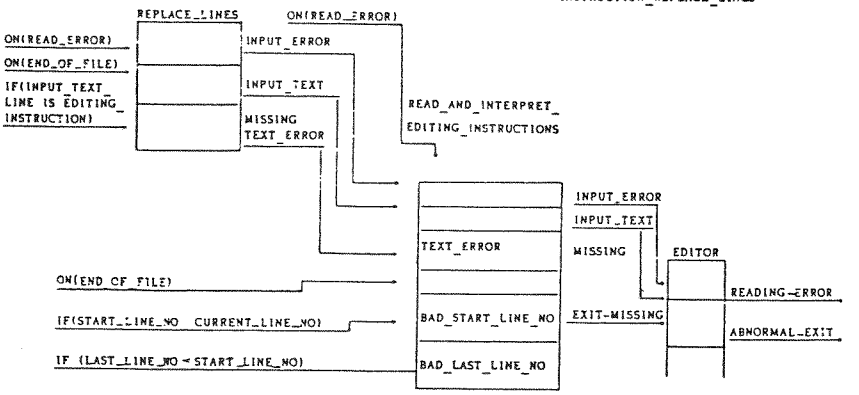
Anche se il formato grafico è dipendente dal tipo di informazione che esse contengono, le check-list sono utili come supporto di documentazione ausiliario.

CONDIZIONE DI INPUT	T.	CONDIZIONE DI OUTPUT	T. AZIONE
ON(INPUT_ERROR.READ_AND_INTERPRET_EDITING_INSTRUCTIONS)		READING_ERROR	F MESSAGGIO:"ERRORE etc." CHIUDI FILE
ON(EXIT_MISSING.READ_AND_INERPRET_EDITING_INSTRUCTIONS)	F	ABNORMAL_EXIT	F MESSAGGIO:"FINE DEL FILE etc.",CHIUDI FILE
ON(INPUT_TEXT_MISSING.READ_AND_INTERPRET_EDITING_INSTRUCTION)	F	i.d.c.s.	F i.d.c.s.

- Condizioni di eccezione: unità EDITOR

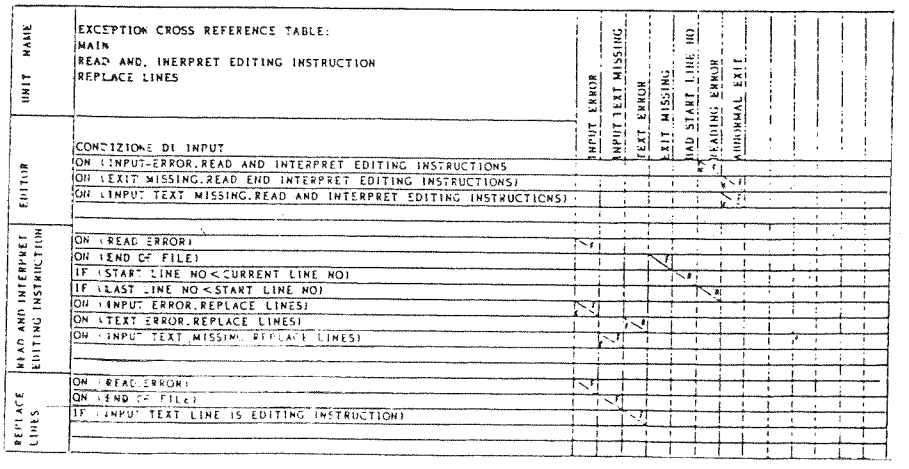
DIAGRAMMA DI FLUSSO DI ECCEZIONI

UNITA': EDITOR
READ_AND_INTERPRET_EDITING_
INSTRUCTION_REPLACE_LINES



Esempio di diagramma di flusso di eccezioni

CONDIZIONE DI OUTPUT



Esempio di tabella di riferimenti incrociati fra eccezioni

Fig. 8 Check list

2.2.3. Diagrammi di Warnier.

I diagrammi di Warnier sono gerarchici e strutturati in due tipi complementari per la descrizione delle funzioni e dei dati. Entrambe le rappresentazioni sono formate da parentesi graffe innestate (e sono traducibili in diagrammi strutturati).

Nei diagrammi funzionali sono introdotte ulteriori convenzioni:
- numeri tra parentesi definiscono la ripetitività di un modulo (v. figg. 9a..c);
- il simbolo "(●)" indica moduli mutuamente esclusivi (v. fig. 9b);
- il simbolo "+" esprime il parallelismo di procesi (v. fig. 9c).

I diagrammi dati, a parte la diversa forma grafica (v. fig. 9a), sono analoghi ai diagrammi di Jackson.

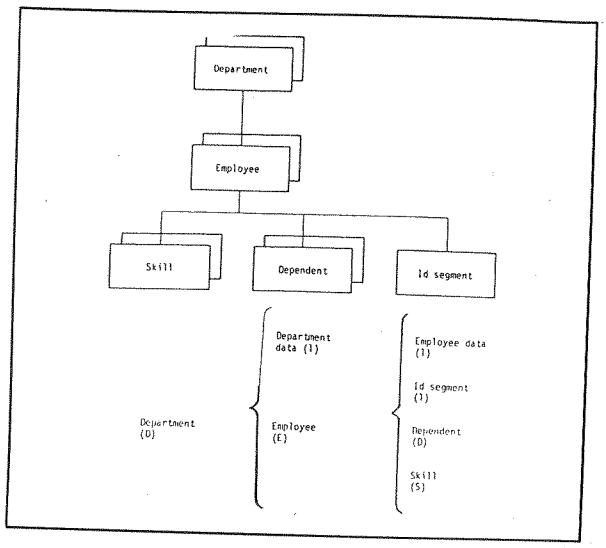


Fig. 9a Diagramma Dati di Warnier

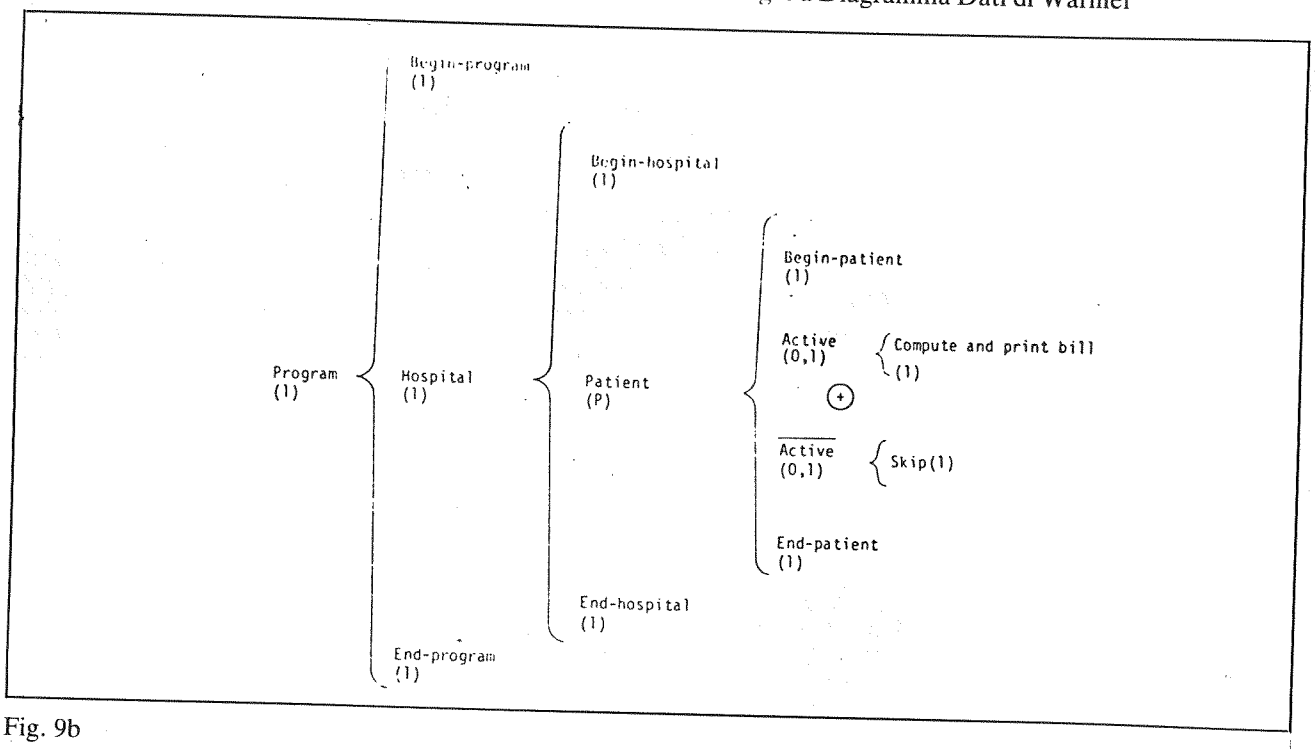


Fig. 9b

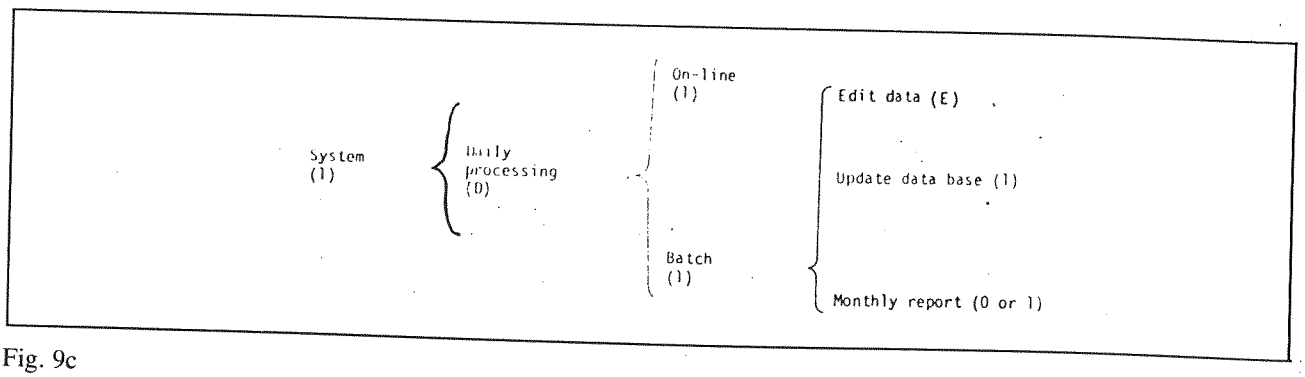


Fig. 9c

2.2.4. Tavole decisionali.

Le tavole decisionali sono suddivise in due parti: "decisione" ed "azione" (v. fig. 10). la parte "decisione" contiene l'elenco delle varie condizioni elementari ("stati") del sistema (o modulo) seguite dall'elenco delle possibili configurazioni di stati. In corrispondenza di ogni configurazione è individuata univocamente la serie di azioni.

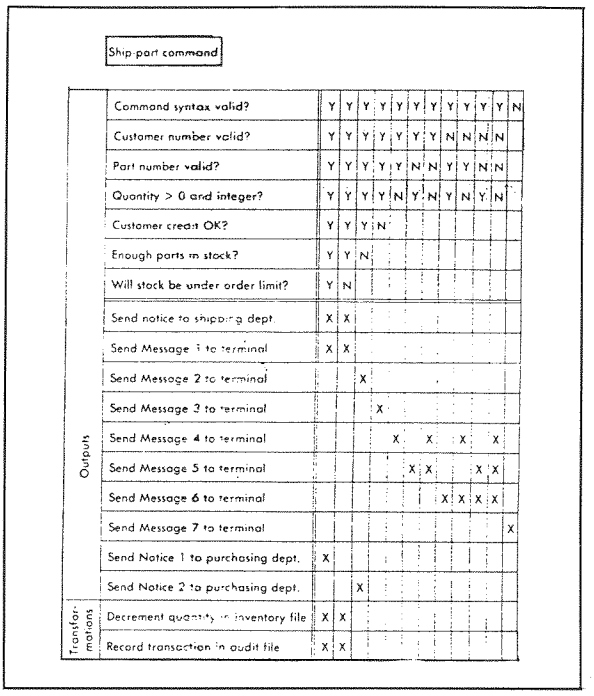


Fig. 10 Tavola decisionale

Le tavole decisionali sono uno strumento valido nella analisi del flusso di controllo, specie in situazioni (espressioni logiche) complesse, anche se oneroso al crescere del numero di stati.

Esse non permettono una rappresentazione dinamica del flusso di controllo, e non descrivono il flusso dei dati.

2.2.5 Riepilogo strumenti semigrafici

Il riepilogo delle esigenze di disegno risolte dagli strumenti semigrafici in considerazione è tracciato in tabella C.

Si noti che i diagrammi di Warnier coprono gran parte delle esigenze (in particolare del software EDP), anche se a differenti livelli di accuratezza.

2.3. Rappresentazioni reticolari.

Le rappresentazioni reticolari sono idonee alla descrizione di vincoli di concorrenza.

I diagrammi reticolari esaminati nei paragrafi seguenti sono:

- diagrammi di stato;
- R-net;
- reti di Petri;
- reti di Petri procedurali;
- AP net.

ESIGENZE / STRUMENTI	ENTITÀ/ RELAZIONI	PROCESSI/ AZIONI	INTERFACCE EST. (UTENTE)	DEFINIZIONE FUNZIONI	FLUSSO DI CONTROLLO	STRUTTURE DATI	FLUSSO DATI	STATI DI PROCESSO	INTERFACCE (MODULI/PROC)	SINCRONIZZAZIONE	VINCOLI TEMPORALI	COMUNICAZIONI ESTERNE	COMUNICAZIONI INTERNE	NOTE
N^2 CHARTS			I	$I/_I$	$S/_I$				$S/_S$	$S/_S$		X	XX	
CHECK-LIST			I	$I/_I$				$I/_I$	$S/_S$			X	XX	VARI USI
DIAGRAMMI DI WARNIER			I	$I/_I$	$F/_F$	$F/_F$	$I/_I$		$I/_I$	$I/_I$		X	XX	LCS?
TAVOLE DECISIONALI			S					$F/_F$				X	XX	

TABELLA C - Esigenze di disegno soddisfatta da strumenti "semigrafici"

2.3.1. Diagrammi di stato.

I diagrammi di stato sono l'esempio più semplice di rappresentazione reticolare e descrivono il funzionamento di un automa a stati finiti (v. fig. 11). Da ogni stato (rappresentato mediante un cerchio) partono tante frecce quanti sono i possibili casi che richiedono azioni diverse; ogni freccia è descritta dal caso ad essa associato e dalla indicazione della legittimità della sequenza.

Le stesse informazioni di un diagramma di stato possono essere contenute in una tabella decisionale.

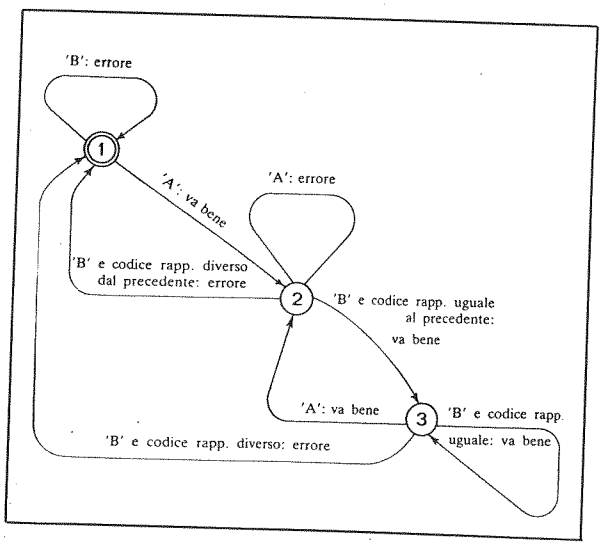


Fig. 11 Diagramma di Stato

2.3.2 R-net (Requirement net).

Le "Requirement net" sono un'evoluzione dei diagrammi a blocchi mediante una simbologia grafica specifica per descrivere aspetti di parallelismo e sincronizzazione (v. fig. 12).

Le R-net consentono la descrizione funzionale delle risposte del sistema a particolari stimoli attraverso la definizione della sequenza delle transazioni da eseguire per generare cambiamenti nello stato del sistema e risposte all'ambiente.

La struttura di una R-net consiste di nodi ed archi che li congiungono. I nodi sono di vari tipi:

- transizioni funzionali;
- subnet (strutture di flusso di livello inferiore);
- punti di validazione, in cui raccogliere dati per misurare le prestazioni del sistema;
- eventi (nodi che abilitano altre R-net);
- nodi di interfaccia di Input/Output;
- nodi "OR" e "FOR EACH", che esprimono le strutture di controllo basilari;
- nodi "AND", che rappresentano il parallelismo tra attività.

Le R-net sono uno strumento di descrizione formale degli aspetti funzionali del sistema.

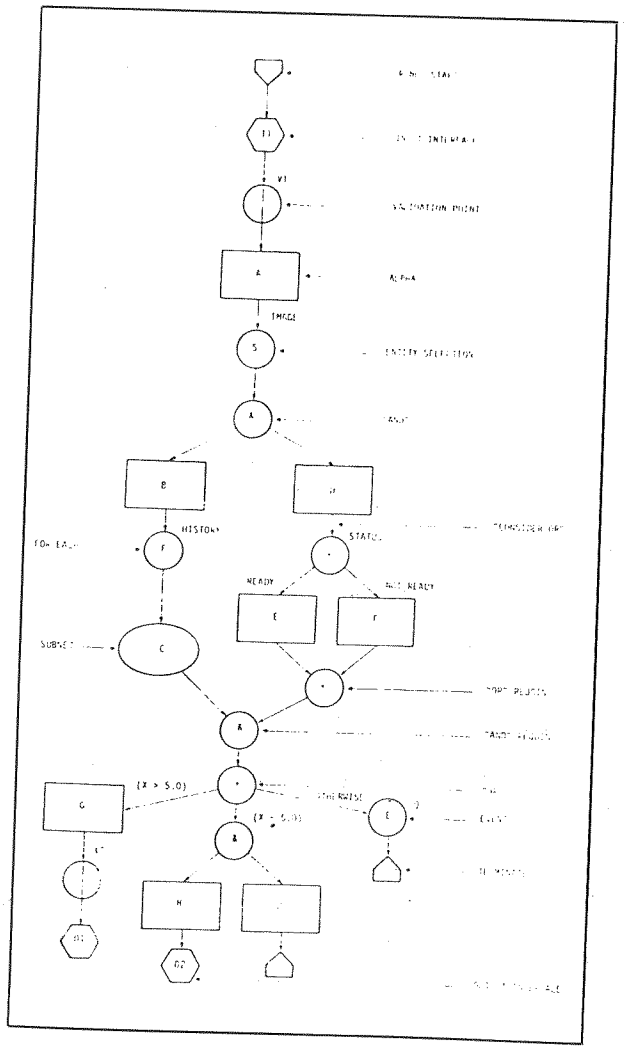


Fig. 12 R NET

2.3.3. Reti di Petri

Le reti di Petri sono un modello formale e grafo di rappresentazione "dinamica" del flusso informativo e di controllo di sistemi.

Una rete di Petri contiene due tipi di nodi: i "posti" (solitamente rappresentati da cerchi), che rappresentano gli stati, e le "transizioni" (solitamente rappresentate da barre) tra stati (v. fig. 13a).

Un punto ("marca") in un posto indica che la condizione è soddisfatta. Una distribuzione di marche ("marcatura") rappresenta lo stato del sistema.

Per modellare il comportamento dinamico di un sistema, l'esecuzione di un processo è rappresentata dallo "scatto" della transizione corrispondente (v. fig.

13b). I cambiamenti nello stato del sistema sono rappresentati dai movimenti delle marche nella rete, secondo le seguenti regole:

1 – una transizione è abilitata se e solo se ciascuno dei suoi posti di ingresso ha almeno una marca;
2 – una transizione può scattare solo se è abilitata;
3 – quando una transizione scatta, viene rimossa una marca da ciascuno dei suoi posti di ingresso, e viene inserita una marca in ciascuno dei suoi posti di uscita.

La "vita", la "limitatezza" e la "corretta terminazione" del modello a rete di Petri definiscono importanti proprietà di un sistema.

Una rete di Petri è "viva" se esiste, per qualunque stato, una sequenza di scatti che abiliti ciascuna transizione della rete. Se una rete di Petri è viva, il sistema non ha punti morti (deadlock).

Una rete di Petri è "limitata" se, per ciascun posto, esiste un limite superiore al numero di marche che vi possono coesistere. La limitatezza di una rete di Petri (numero massimo di eventi presenti in un posto) può essere correlata a limiti superiori di risorse (ad. es. buffer di memoria).

Se il limite superiore sul numero di marche in ciascun posto è uno, la rete di Petri è "sicura". I costrutti di programmazione tipo "regioni critiche" e "monitors" possono modellarsi tramite reti di Petri sicure.

Una rete di Petri è a "corretta terminazione" se termina senza marche nella rete (il sistema non presenta effetti collaterali alla successiva partenza).

Una marcatura definisce un insieme di transizioni abilitate. Se tale insieme è vuoto, l'esecuzione della rete di Petri si arresta.

Due transizioni abiltate sono dette "concorrenti" (parallele) se i loro posti d'ingresso sono disgiunti; altrimenti esse sono abilitate contemporaneamente, deve essere effettauata una scelta discrezionale. (v. fig. 13c).

La modellizzazione di un sistema con l'impiego (interpretato a seconda delle esigenze) delle reti di Petri ha tre vantaggi principali:

. la rappresentazione grafica è formale ed estremamente sintetica;

. la teoria delle reti di Petri, anche se presenta aspetti di complessità matematica, comprende strumenti di analisi (quali gli alberi di marcatura e gli invarianti di rete, e relazioni tra strutture di rete ed il loro comportamento dinamico) utili nell'esame del comportamento del sistema;

. le reti di Petri possono essere sintetizzate con approccio bottom-up e/o top-down.

Una rete rappresenta un sistema quando viene assegnata un'interpretazione alle varie entità (i posti, le marche e le transizioni); all'interno dello stesso campo applicativo è possibile utilizzare diversi livelli di interpretazione, il cui grado dipende dal tipo di informazione richiesta.

Sulla base dell'interpretazione fornita, le reti di Petri possono essere utilizzate per rappresentare i sistemi a vari livelli di astrazione e di dettaglio. Ad esempio, facendo corrispondere ad ogni marca un insieme di dati, si definisce il flusso di dati nella rete.

Uno sviluppo delle reti di Petri include la descrizione del fattore tempo, e quindi consente di definire vincoli temporali.

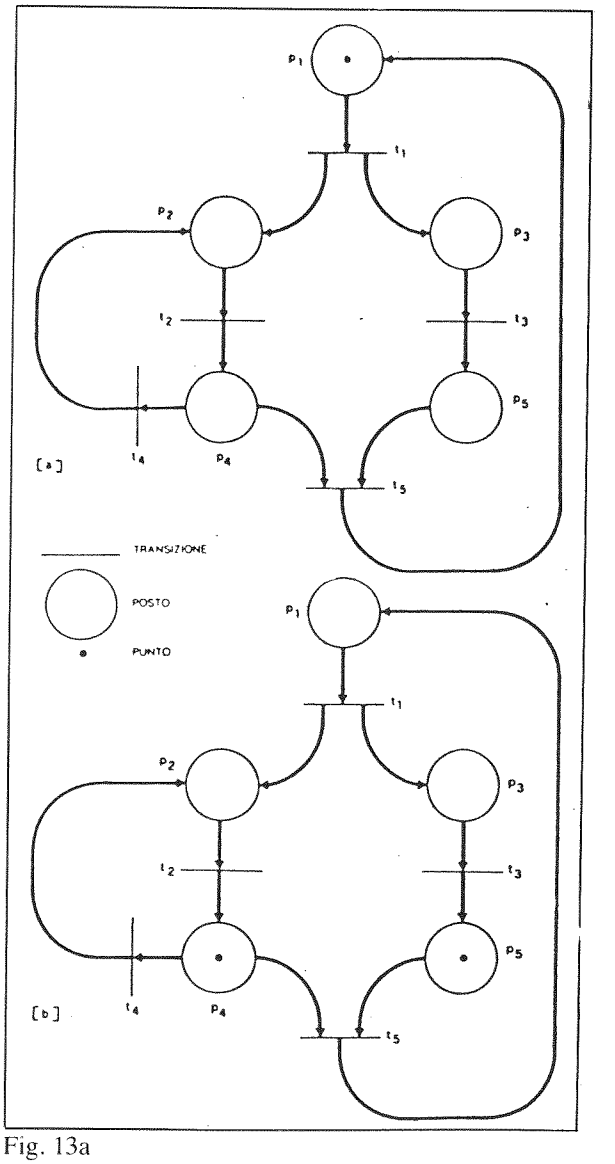


Fig. 13a

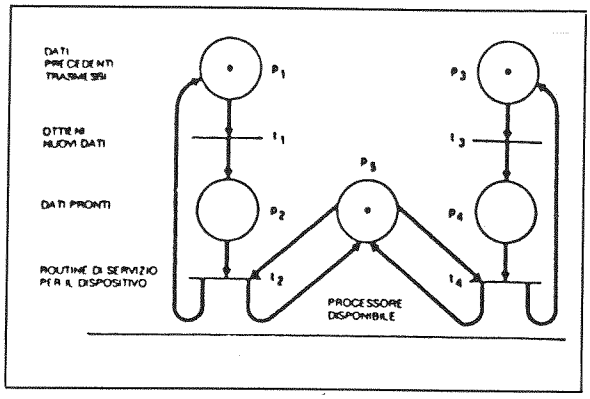


Fig. 13b Rete di Petri

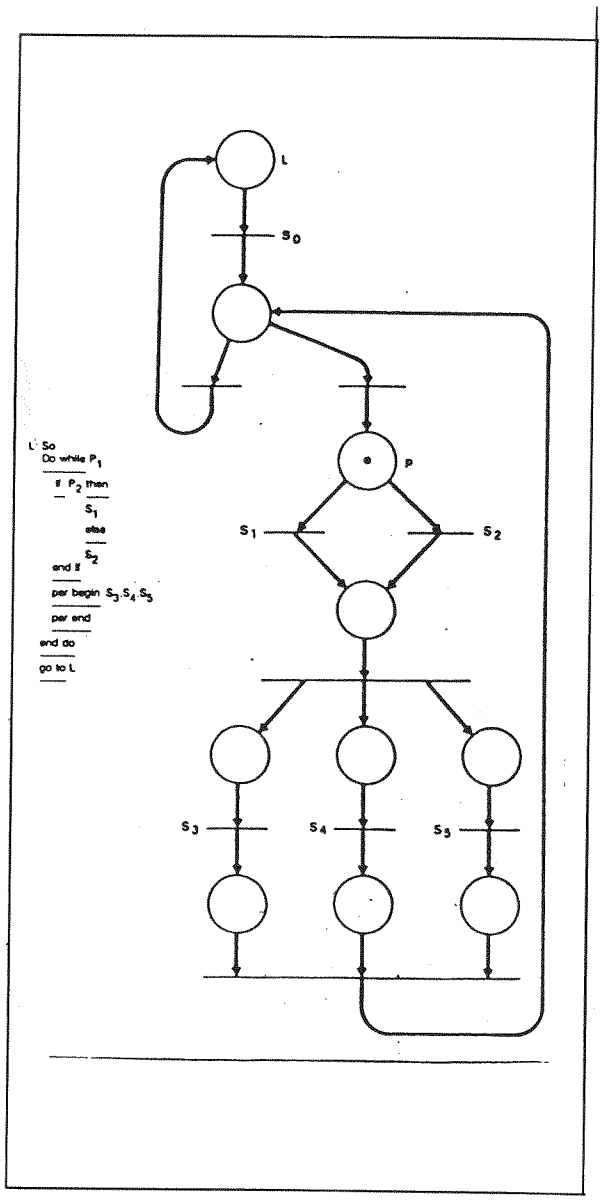


Fig. 13c Rete di Petri

2.3.4. Reti di Petri Procedurali.

Le reti Procedurali (v. studi dell'Istituto di Cibernetica dell'Università di Milano) propongono le seguenti estensioni alle reti di Petri:

- interpretazione di una rete in termini di RISORSE (posti) / ATTIVITÀ (transizioni);
- rappresentazione grafica orientata alla descrizione di procedure: le risorse sono indicate da simboli grafici memonici (dischi, listing, etc.) e le attività sono indicate in rettangoli (v. fig. 14a);
- notazioni modificate e abbreviate per rappresentare operazioni di AND/OR (v. fig. 14b).

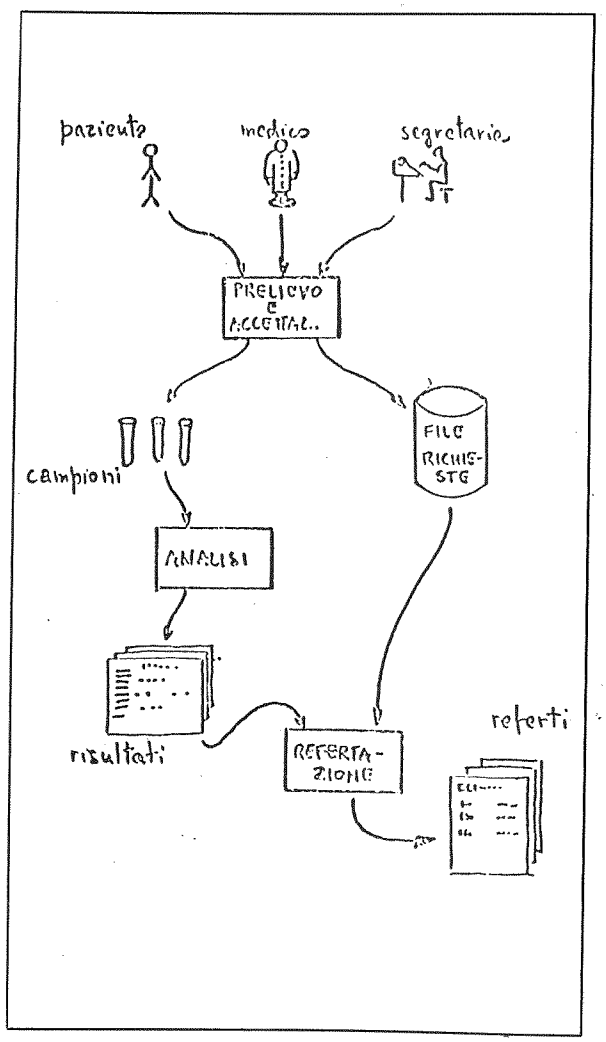


Fig. 14a Rete di Petri procedurale

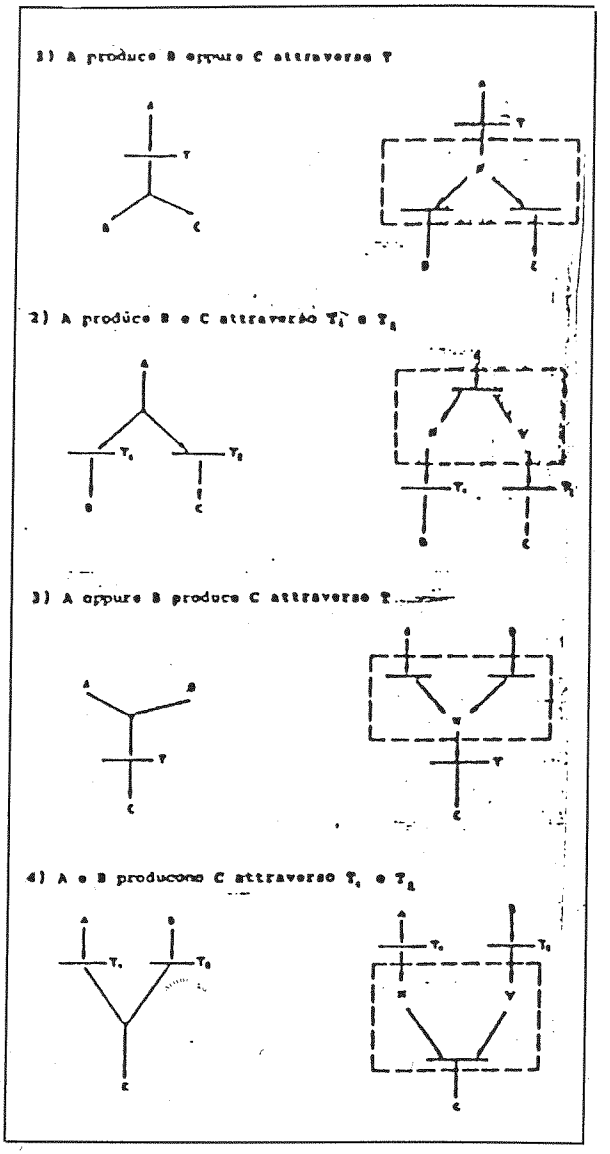


Fig. 14b

2.3.5 AP-net.

Le "Abstract Process Network" sono una variante delle reti di Petri "sicure", estese mediante l'introduzione dei concetti di "processo astratto" (AP) e "transizione a definizione di stato".
Gli obiettivi dichiarati dalle AP-net sono orientati alla definizione di "software-blueprint" che posseggano i seguenti requisiti:

- rappresentare la struttura logica di un sistema e le corrispondenti trasformazioni dei dati a livelli diversi di astrazione;
- fornire una speficica completa e non ambigua del sistema da usare nelle varie fasi del ciclo di vita;
- fornire un semplice mezzo di espressione grafica;
- fornire una semplice notazione algebrica per la rappresentazione del disegno, convertibile in una forma grafica equivalente (per facilitare l'analsi, la modifica e la verifica del disegno per mezzo di operazioni algebriche formali);
- stimolare un processo di disegno disciplinato affinchè il progetto risulti organizzato gerarchicamente, modulare e strutturato;
- fornire un semplice mezzo per la modellizzazione delle proprietà dinamiche di un sistema identificando esplicitamente le relazioni tra le varie condizioni, decisioni ed azioni;
- integrare i principi base della progettazione di sistemi (astrazione, analisi, sintesi, docomposizione funzionale, gerarchia, modularità, etc.) per una loro efficace applicazione durante il processo di disegno del software.

Un processo astratto è formalmente definito come una quintupla P=<A,I,F,O,Z> in cui A è lo stato iniziale, I è l'input, F la funzione O l'output e Z lo stato finale. Il flusso di controllo A ⇒F⇒ Z è ortogonale al flusso dei dati I ⇒F ⇒O attraverso la funzione (v. fig.. 15a).

L'algebra dei processi astratti si basa su poche regole principali (corrispondenti ad equivalenti forme grafiche) che sono basate sui concetti delle strutture di controllo fondamentali ("Dijkstra-charts"):

1) 'P=ab' indica una decomposizione sequenziale del processo 'P' nel processo 'a' seguito dal processo 'b' tale che:
A(P)=A(a); Z(P)=Z(b); Z(a)<=A(b).

2) 'P=a+b' indica una decomposizione selettiva di P in 'a' o 'b' tale che: A(P)=A(a)∨A(b); A(a)∧A(b)=O;

3) 'P'=a*b' (a<>O, b<>O) è uno schema iterativo, dove 'a*' indica una sequenza finita di n occorrenze (n>=O) del processo 'a' seguita da un'occorenza del processo 'b'.

4) 'P = alb indica la concorrenza dei processi 'a' e 'b' tale che ab = ba.

5) 'P=aP+b' è uno schema ricorsivo del processo astratto P rappresentato in termini dei processi componenti 'a' e 'b'.

6) 'a ^'indica un processo identico, cioè tale che a^ = <A=Z<>O,F=I=O=O>, e 'O' indica un processo nullo, tale che: a+O=a, Ob=bO= =O, O*b=b.

Le AP-nets introducono le strutture di controllo di base ed eliminano le ambiguità interpretative che si verificano in situazioni di conflitto nelle reti di Petri.

Le strutture di controllo di sequenza, selezione, iterazione, ricorsività e concorrenza sono rappresentate secondo gli schemi delle figg. 15b,c.

Le transizioni funzionali sono rappresentate da rettangoli ed hanno un solo posto di ingresso ed un solo posto di uscita, ai quali sono associati dei predicati sullo stato del sistema detti pre-condizione e post-condizione.

La logica decisionale è espressa da operatori logici (AND,OR) che agiscono sui predicati di pre/post-condizione.

L'insieme T delle transizioni è suddivisibile in un sottoinsieme D di transizioni a definizione di stato ed in un sottoinsieme F di transizioni di trasformazione (la relazione tra i posti e le transizioni è definita da un insieme di marcature M). Tutti i posti e le transizioni sono definiti rispetto ad un vettore V di variabili di stato del sistema. Perciò una AP-net N è rappresentabile tramite una quadrupla N=<P,T,M,V>.

Le AP-nets sono libere da conflitti in quanto ciascun posto può avere una sola transizione in ingresso ed una sola uscita. Al più alto livello di astrazione una AP-net consiste di una sola transizione funzionale che elimina il concetto di marcatura iniziale (la sua esecuzione ha inizio quando il posto di input riceve una marca, particolare insieme di valori delle variabili di stato).

Il numero totale di marche durante l'esecuzione di una AP-net (non necessariamente costante) indica il numero di processi concorrenti attivi.

La stesura di una AP-net ha origine da un diagramma di stati-transizioni (v. fig. 15d), dal quale si eliminano successivamente i conflitti introducendo le transizioni a definizione di stato (v. fig. 15e) (ogni stato è ridotto ad una sola transizione di ingresso ed una sola di uscita).
Infine la struttura viene corretta eliminando processi in comune mediante duplicazione e con l'introduzione di ulteriori transizioni a definizione di stato (v. fig. 15f).

Una AP-net è esprimibile anche in un'espressione regolare a partire dalla matrice di connettività "C" (v. fig. 15g).

Le AP-nets possono essere anche sintetizzate bottom-up riunendo (ricorsivamente) in una transizione a definizione di stato tutte le transizioni funzionali che partono da uno stato (v. fig. 15h).

Data l'ortogonalità dei flussi di controllo e dei dati nella definizione di un processo astratto è possibile, per lo stesso sistema, costruire due AP-nets, l'una per il flusso di controllo e l'altra per il flusso dei dati.

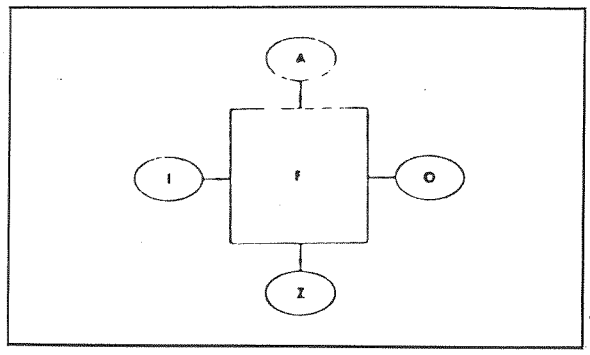


Fig. 15a

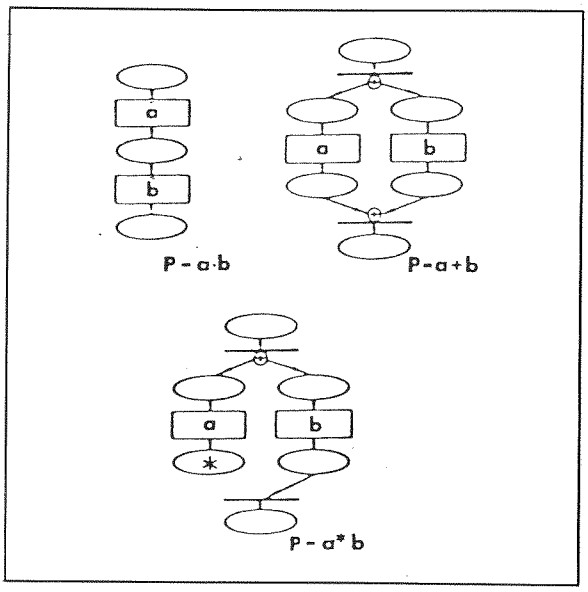


Fig. 15b Notazioni delle AP NET

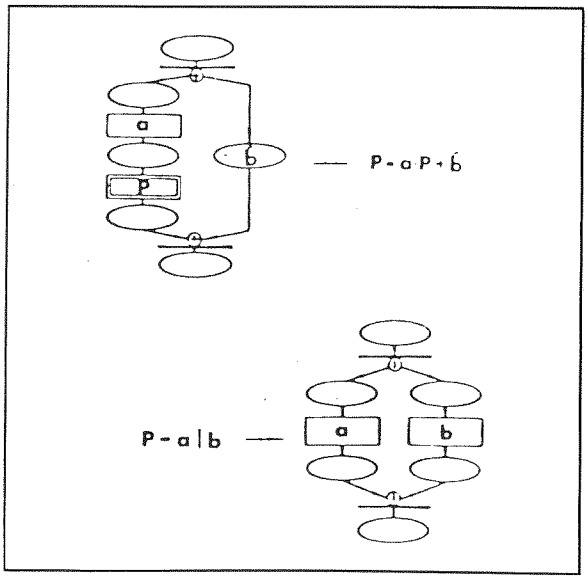


Fig. 15 c

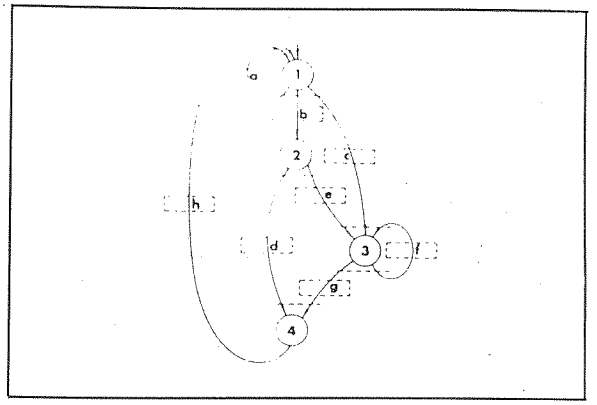


Fig. 15d

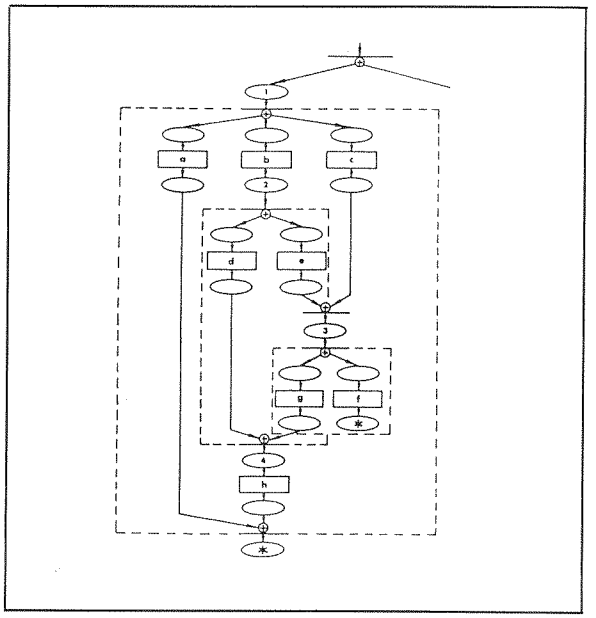


Fig. 15e Procedura di stesura di una AP NET

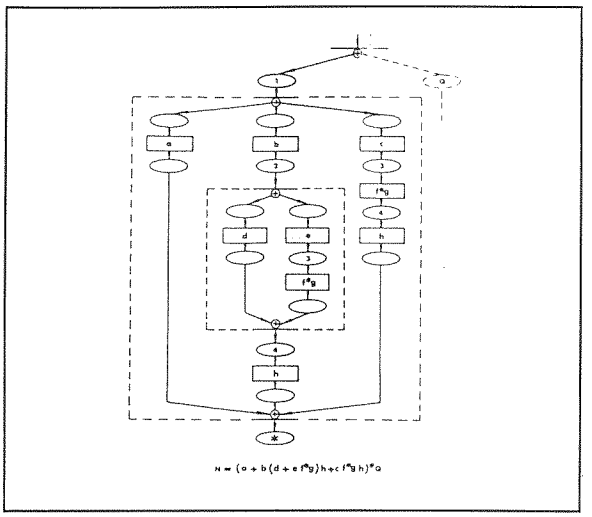


Fig. 15f

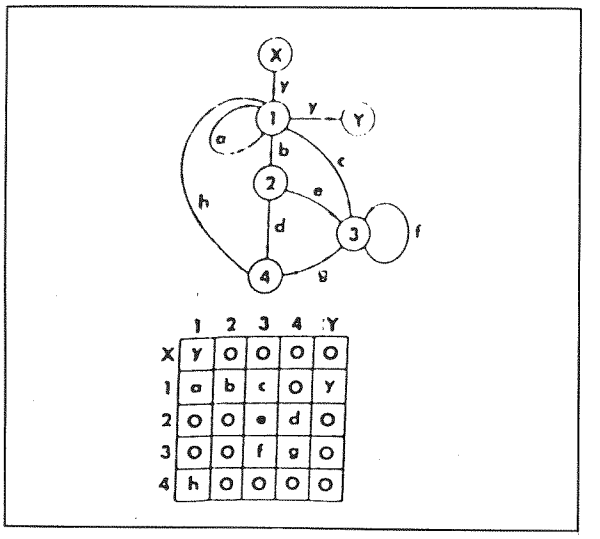


Fig. 15g Matrice di connettività

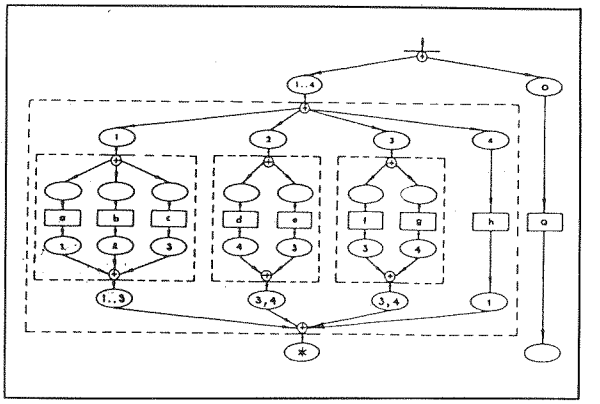


Fig. 15h Sintesi di AP NET

2.3.6. Riepilogo delle rappresentazioni reticolari.

La tabella D riassume la valutazione della capacità di rappresentazione degli aspetti software da parte di diagrammi reticolari.

Da tale valutazione appare che le rappresentazioni reticolari (ed in particolare quelle basate sulle reti di Petri) sono idonee a descrivere formalmente molteplici aspetti (in particolare i vincoli di sincronizzazione e di prestazione scarsamente definiti dagli altri strumenti grafici).

Le reti di Petri Procedurali propongono una forma grafica particolarmente semplice ed idonea quale mezzo di comunicazione con il committente.

Le AP-nets presentano il pregio della formalizzazione (corrispondenza reticolo-espressione regolare), che consente una rigorosa descrizione della maggioranza degli aspetti software e quindi costituisce anche un valido strumento di comunicazione all'interno dei gruppi di lavoro.

ESIGENZE / STRUMENTI	ENTITÀ/ RELAZIONI	PROCESSI/ AZIONI	INTERFACCE EST. (UTENTE)	DEFINIZIONE FUNZIONI	FLUSSO DI CONTROLLO	STRUTTURE DATI	FLUSSO DATI	STATI DI PROCESSO	INTERFACCE (MODULI/PROC)	SINCRONIZZAZZIONE	VINCOLI TEMPORALI	COMUNICAZIONI ESTERNE	COMUNICAZIONI INTERNE	NOTE
DIAGRAMMI DI STATO			I		S/S			F/F		F/F		XX	X	
R-NET		F	S	I/I	F/F		S/S	F/F	S/S	F/F	F/F	X	XX	
RETI DI PETRI		F	I	I/I	S/S		F/F	F/F		F/F	F/F	X	XXX	
RETI PROCEDURALI	I	F	I	I/I	F/F		F/F	F/F		F/F	F/F	XXX	XXX	
AP NET	I	F	I	I/I	F/F		F/F	F/F		F/F	F/F	X	XXX	

TABELLA D – Esigenze di disegno soddisfatte da rappresentazioni reticolari

3. VALUTAZIONI CONCLUSIVE

Le rappresentazioni grafiche basate sulle reti di Petri sono idonee a descrivere a livello formale una gran parte degli aspetti software.

I diagrammi E-R coprono a livello formale aspetti complementari.

Dall'integrazione diagrammi reticolari/diagrammi E-R non sono coperti a livello formale (e da nessuno altro strumento grafico) solo tre aspetti:

- interfacce esterne ed in particolare interfacce con l'utente;
- definizione delle funzioni;
- definizioni delle interfacce tra moduli e/o processi.

È evidente che questi tre aspetti presentano specifiche peculiarità. Le interfacce con l'utente possono essere descritte formalmente tramite linguaggi di descrizione orientati, corredati da rappresentazioni grafiche (tipicamente formati video, tracciati di stampa).

Le funzioni sono descrivibili efficacemente da linguaggi funzionali formali o semiformali, quali ad es. WELLMADE, CCS (Calculus of Communicating Systems), SPECIAL dell'HDM (Hierarchical Development Methodology), VDM (Vienna Development Methodology), etc.

Gli aspetti di interfaccia tra moduli/processi possono essere trattati probabilmente con un approccio tipo E-R (Entità = moduli o processi, Relazioni = dati di interfaccia), integrate da un dizionario dei dati.

Un set di strumenti comprendente:

- diagrammi reticolari di tipo Procedural Net o AP Net;
- diagrammi E-R, probabilmente idonei anche alla descrizione delle interfacce fra moduli e/o processi;
- un linguaggio di descrizione di interfaccia utente-sistema;
- un linguaggio funzionale (formale o almeno semiformale);

dovrebbe pertanto essere idoneo a soddisfare tutte le esigenze di disegno.

4. BIBLIOGRAFIA

(1) *"Definizione di Linguaggi di Specifica e Descrizione"*, CSELT, 1980.

(2) *"ADA-based System Development Methodology Study Report"*, UK Department of Industry, 1981.

(3) *"Il Processo di Produzione SW – Standard – Parte 1 – Introduzione e Principi Ispiratori"*, PFI – METOD, 1979.

(4) *"Giornata di lavoro su: l'Uso delle Reti di Petri nell'Ingegneria del Software"*, PFI – METOD, 1980.

(5) *"Giornata di lavoro su: Problemi di definizione di Requirements, di Analisi e di Disegno di Sistemi Software"*, PFI – METOD, 1979.

(6) L.J. Mekley, S.S. Yau: *"Software Design Representation Using Abstract Process Networks"*, IEEE Trans. SE Vol. 6, pp. 420-433. Sept. 1980.

(7) G. Castelli, F. De Cindio, G. De Michelis, G, Haus, M. Maiocchi, C. Simone: *"Verso una metodologia per la costruzione di specifiche strutturate e corrette di procedure e programmi"*, Atti AICA 1979, Vol. 2.

(8) Z. Manna: *"Mathematical Theory of Computation"*, New York, Mc Graw-Hill, 1974.

(9) R.C. Tausworthe: "Standardized Development of Computer Software", Englewood Cliffs, NJ, Prentice-Hall, 1977.

(10) Infotech State of the Art Report: *"Software Engineering Techniques"*, Voll. 1 e 2, 1977.

(11) Infotech State of the Art Report: *"Structured Analysis and Design"*, Voll. 1 e 2, 1978.

(12) D.T. Ross: *"Structured Analysis (SA): A language for communicating ideas" IEEE Trans. SE Vol. 3, pp. 16-34, Jan. 1977.*

(13) "SADT – Structured Analysis & Design Technique – Reader Course", Softech Inc., 1976.

(14) A.C. Shaw: *"Software descriptions with flow expressions"*, IEEE Trans. SE vol. 4, pp. 242-254, May 1978.

(15) H.A. Sholl, T.L. Booth: *"Software performance modelign using computation structures", IEEE* Trans. SE Vol. 1, pp. 414-420, Dec. 1975.

(16) U.R. Kodres: *"Analysis of real-time system by data-flowgraphs"*, IEEE Trans. SE Vol. 4, pp. 169-178, May 1978.

(17) E.W. Dijkstra: *"Notes on structured programming"* in Structured programming, O.J. Dalh, New York, Academic 1972, pp. 1-81.

(18) U.R. Kodres: *"Discrete systems and flowcharts"*, IEEE Trans. SE Vol. 4, pp. 521-525, Nov. 1978.

(19) M.A. Jackson: *"Principles of Program Design"*, New York, Academic Press 1975.

(20) M.A. Jackson: *"J.S.D."*, Michael Jackson Associates, 1981.

(21) M.A. Jackson: *"Information Systems: Modelling Sequencing and Transformations"*, Proc. 3rd Int. Conf. on Software Engineering, pp. 72-81, 1978.

(22) J.D. Warnier: *"Logical construction of programs"*, New York, Van Nostrand Reinhold, 1976.

(23) J.D. Warnier: *"Precis de logique Informatique. L'Organisation des donnes d'un systeme (L.C.S.)"*, Les Edition d'Organisation, Paris, 1980.

(24) M.R. Paige: *"Program graphs, an algebra, and their implications for programming"*, IEEE Trans. SE Vol. 1, pp. 286-291, Sept. 1975.

(25) J.L. Peterson: *"Petri nets"*, ACM Computing Surveys Vol. 9, pp. 223-252, Sept. 1977.

(26) J.D. Noe, G.J. Nut: *"Macro E-nets for representazion of parallel systems"*, IEEE Trans. Comput. Vol. 22, pp. 718-727, Aug. 1973.

(27) C. Davis, C. Vick: *"The software development system"*, IEEE Trans. SE Vol. 3, pp. 69-84, Jan. 1977.

(28) J.L. Baer, C.S. Ellis: *"Model, design and evalutation of a compiler for a parallel processing environment"*, IEEE Trans. SE Vol. 3, pp. 394-405, Nov. 1977.

(29) C.A.R. Hoare: *"An axiomatic basis for computer programming"*, Commun. A.C.M. Vol. 12, pp. 576-580,583, Oct. 1969.

(30) E.W. Dijkstra: *"Guarded commands, nondeterinacy and formal derivation of programs"*, Commun. A.C.M. 18, pp. 453-457, Aug. 1975.

(31) E.W. Dijkstra: *"A discipline of programming"*, Englewood Cliff, NJ, Prentice-Hall, 1976.

(32) D. Gries: *"An illustration of current ideas on the derivation of correctness proof and correct programs"*, IEEE Trans. SE Vol. 2, pp. 238-244, Dec. 1976.

(33) P. Chung, B. Gaiman: *"Use of state-diagrams to engineer communications software"* in Proc. III Int. Conf. Software Eng., pp. 215-221, May 1978.

(34) T.S. Chow: *"Testing software design modeled by finite-state machines"* IEEE Trans. SE Vol. 4, pp. 178-187, May 1978.

(35) D.L. Boyd, A. Pizzarello: *"Introduction fo the WELLMADE design methodoloty"*, IEEE Trans. SE Vol. 4, pp. 276-282, July 1978.

(36) J. Backus: *"Can programming be liberated from the Von Neumann style? A functional style and ist algebra of programs"*, Commun. A.C.M. Vol. 21, pp. 613-641, Aug. 1978.

(37) F. Capozza, C. Di Noi, C. Gallo, F. Esposito, A. Lanza: *"Il trattamento delle eccezioni nella programmazione strutturata: metodologia di progetto"*, Congresso Annuale AICA 1981, Atti Vol. 2.

(38) P.P. Chen: *"The Entity Relationship Model: Toward a Unified View of Data"*, A.C.M. Trans. on Database System, vol. 1, March 1976.

(39) P.P. Chen: *"Entity Relationship Approach to System Analysis and Design"*, North Holland, 1980.

(40) H. Bratman: *"The Software Factory"*, Computer Vol. 8, pp. 28-37, May 1975.

(41) B.W. Boehm, R.K. Mc Clean, D.B. Urfrig: *"Some Experience with Automated Aids to the Design of Large-Scale Reliabe Software"*, IEEE Trans. SE Vol. 1, pp. 125-133, Jan. 1975.

(42) J.D. Cougar: *"Evolution of Business System Analysis Techniques"*, Computing Surveys Vol. 5. pp. 167-198, Mar. 1973.

(43) P. Freeman: *"Automating Software Design"*, Computer Vol. 7, pp. 33-38, Apr. 1974.

(44) O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare: *"Structured Programming"*, London, Academic Press, 1972.

(45) C.B. Jones: *"Formal Definition in Program Development"* in C.E. Hackl Ed., Programming Methodology, Berlin, Springer-Verlag, pp. 387-443, 1975.

(46) B.H. Liskov: *"A Design Methodology for Reliable Software Systems"*, Proceedings of the 1972 Fall Joint Computer Conference, Montvale, NJ, AFIPS Press 1972, pp. 191-199.

(47) D.L. Parnas: *"On the Criteria to be Used in Decomposing Systems into Modules"*, Commun. A.C.M. Vol. 15 (2), pp. 1053-1058, 1972.

(48) E. Yourdon, L.L. Constantine: *"Structured Design"*, New York, 1975.

(49) HIPO – *A Design Aid and Documentation Techniqhe*, GC20-1851, IBM Corp., White Plains, NY, 1974.

(50) M.L. Hughes, R.M. Shank, E.S. Steins: *"Decision Tables"*, New York, Mc Graw-Hill, 1968.

(51) G.J. Myers: *"Software Reliability: Principles and Practices"*, John Wiley & Sons.

Nota: Il presente lavoro è apparso negli Atti del Convegno su "Metodologie di analisi e di disegno nello sviluppo di software industriale", Milano FAST, 27-28-29 giugno 1983, organizzato nell'ambito del Progetto Finalizzato per l'Informatica del CNR, Sottoprogetto P1, obiettivo METOD.

L'APPROCCIO DI JACKSON AL SYSTEM DEVELOPMENT (JSD): ALCUNE NOTE.

Carla Simone
Istituto di Cibernetica – Università di Milano

Riassunto

Queste note presentano il metodo proposto da M. Jackson per il system development basato sulla costruzione di modelli del sistema da realizzare e dell'ambito in cui dovrà operare. Dopo un suo inquadramento all'interno delle problematiche dell'ingegneria del software vengono delineate le fasi principali che lo costituiscono utilizzando un semplice esempio.
Alcune note critiche mettono in evidenza da un lato gli aspetti innovativi e dall'altro alcune carenze nella modellizzazione della concorrenza, in particolare della mancanza di una base teorica per governare la costruzione dei modelli e la verifica delle loro proprietà e consistenza reciproca.

Dal momento della sua nascita nel 1968, il termine "software engineering" ha assunto diverse caratterizzazioni, che cercavano di volta in volta di specificare che cosa questa disciplina sia, quale funzione abbia o quali strumenti metta a disposizione o utilizzi e via dicendo.

In questo processo di definizione, la primitiva analogia con le altre branche dell'ingegneria, "di essere basata su fondamenti teorici e su discipline applicative" [1] è stata variamente rispettata, toccando di volta in volta i due estremi, quello teorico e quello applicativo, con una generale difficoltà di individuare una giusta via di mezzo, una sintesi accettabile da entrambi i punti di vista e rispetto agli obiettivi di questa ingegneria.

In questa branca dell'ingegneria, più che in altre, è infatti possibile che "una sola disciplina non sia sufficiente per tutti i campi del software, senza essere così generale da essere inutilizzabile per applicazioni pratiche: la teoria può essere generale, la pratica deve essere specifica" [2].

Nel S/E (da questo punto con S/E si abbrevierà "Software Engineering") come nelle altre ingegnerie, il problema è quello di costruire un prodotto a partire da specifiche, in modo che ne siano rispettati, da un lato, tutti i requisiti e, dall'altro, la economicità di costruzione.

In questo passaggio tra specifiche e realizzazione si giocano i problemi essenziali del S/E, sia dal punto di vista del prodotto che da quello della organizzazione che supporta la sua costruzione.

Allora, come specificare? Che cosa significa? Come implementare? Con quali strumenti?

In tutto questo processo il nodo fondamentale sta nella costruzione di modelli, a diversi livelli di astrazione, consistenti l'uno con l'altro, che partendo dal problema conducano ad un modello eseguibile da una macchina.
È quindi sulla capacità di costruire modelli consistenti che si devono concentrare gli sforzi ed è su questo terreno che si può forse trovare la sintesi tra teoria e applicazioni a cui si accennava prima.

Questo tipo di approccio non è forse rivoluzionario, ma la sua forza sta sia nella chiarezza con cui è stato epresso da Jackson in un suo seminario a Milano nella scorsa primavera, sia nella sua capacità di considerare, da un punto di vista applicativo quale è quello dell'autore, i problemi del S/E nella loro essenzialità, come un distillato che focalizza la questione di fondo (che cosa è) dalla quale si ricavano le indicazioni per la risoluzione di altre questioni (come si realizza, con quali strumenti, con quale ambiente di sviluppo, ecc).
Una ingegnerizzazione, dunque, per determinare una struttura produttiva, non per risolvere i suoi problemi. Una ingegnerizzazione che avrà quindi l'esigenza di solide basi metodologiche che inducano una struttura produttiva.

Alcuni approcci metodologici recentemente proposti cercano di raggiungere queste caratteristiche: tra di essi si colloca la proposta JSD (Jackson System Development), illustrata nell'omonimo libro edito da Pretice-Hall [3].

In sintonia con quanto detto precedentemente, due sono le fasi del metodo: la specifica del sistema e la sua realizzazione. Esse hanno compiti diversi, coinvolgono soggetti diversi.

La specificazione ha il compito di definire quali siano le interfacce del sistema da realizzare con la realtà in cui deve essere inserito. Coinvolge quindi direttamente l'utente, le sue conoscenze su tale realtà, la sua esperienza.
La realizzazione invece è un fatto tecnico, che coinvolge specialisti del software, utilizza strumenti automatici di costruzione e in cui la figura dell'utente non è più presenza fondamentale come nella fase precedente.

Come garantire quindi che da parte sua l'utente sia in grado di valutare la adeguatezza delle informazioni fornite e la completezza delle richieste e che, dall'altra parte, il sistema soddisfi i requisiti posti dall'utente stesso?
La risposta si pone, da un lato, nel mettere a disposizione dell'utente un metodo di costruzione dei modelli che lo veda partecipe attivo, critico e creativo e, dall'altro, nel mettere a disposizione di chi realizza concretamente il sistema strumenti di costruzione che garantiscano la consistenza dell'applicazione con le sue specifiche.

Caratteristiche del metodo di costruizione dei modelli devono quindi essere sia la leggibilità e la modificabilità che la formalità (nel senso di non ambiguità del significato) e la capacità di modellare gli aspetti dinamici dell'ambiente e del sistema, cioè le loro interazioni.

Sulla base di questi requisiti gli approcci tradizionali al problema della definizione delle specifiche del sistema sono secono Jackson ampiamente insoddisfacenti, in particolare per quanto riguarda l'ultimo aspetto.
Check-lists e modelli orientati ai dati sono infatti strumenti di modellazione statica: essi separano la descrizione delle funzioni dagli esecutori e dai dati relativi, e non riescono a catturare gli aspetti di interazione tra sistema ed ambiente e di evoluzione del sistema stesso.

Una proposta per ridurre questi inconvenienti è stata presentata in [4], dove l'approccio alle basi di dati viene integrato con descrizioni dinamiche del comportamento del sistema, utilizzando le reti di Petri.
La critica di Jackson non risparmia nemmeno gli approcci gerarchico/funzionali: qui gli aspetti dinamici possono venire modellati, ma il punto critico sta nella riduzione del sistema in oggetto, costituito da componenti umane e componenti automatizzate, ad un unico blocco funzionale iniziale, che viene man mano strutturato in un procedimento top-down.
Tale struttura è però totalmente arbitraria e può causare degli inconvenienti ai livelli di raffinamento inferiori in quanto la scomposizione non rispetta una reale struttura del sistema analizzato, ma risponde unicamente alle esigenze di modellazione dell'osservatore.

È quindi necessario abbandonare quegli strumenti analitici e rivolgersi ad altri più adatti a catturare gli aspetti dinamici e non gerarchici dei sistemi reali. Per Jackson uno di tali strumenti è il concetto di processo sequenziale comunicante, così come è stato introdotto da C.A.R. Hoare nel suo linguaggio dei CSP [5].

Per modellare tali processi il linguaggio proposto in JSD è quello dei ben noti schemi, introdotti da Jackson stesso nel suo metodo di derivazione di programmi JSP [6], per modellare le strutture di dati: lo schema sequenziale, alternativo ed iterativo.
Come modellare gli aspetti di comunicazione in tali schemi? Essi sono evidentemente inadeguati: è quindi necessario introdurre un nuovo linguaggio, questa volta lineare, che può essere assimilato ad un linguaggio Pascal-like con le primitive di read/write su di una coda (data stream) o di getsw su una struttura di dati condivisa (state vector).
Gli schemi precedenti vengono sostanzialmente sostituiti da tale descrizione lineare del controllo in ciascun processo unitamente a nuovi schemi, molto simili ai data flow per descrivere il flusso dei dati tra i processi.

Le figg. 1, 2A e 2B illustrano questi linguaggi in un semplice esempio.

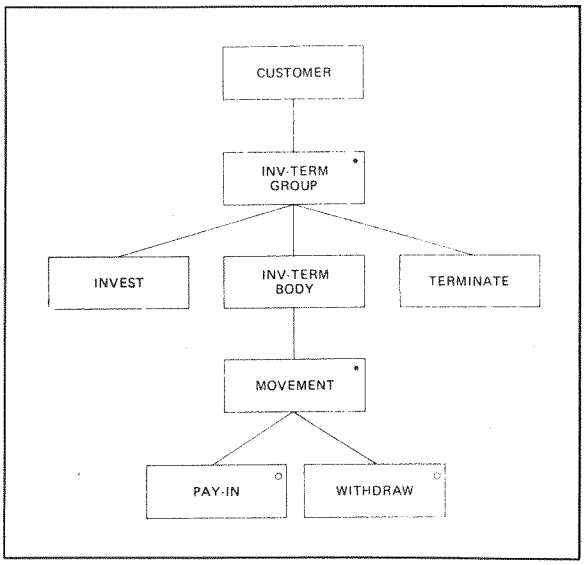


Fig. 1 - Il processo cliente: la sua struttura sequenziale - (Entity structure step)

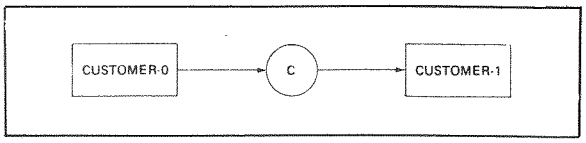


Fig. 2A Due processi cliente: quello reale e la sua immagine nel sistema (Initial model step) Essi comunicano mediante il data stream C.

```
CUSTOMER-1 seq
  read C;
  INVEST; read C;
  CUSTOMER-BODY itr while (PAY-IN or WITHDRAW)
    MOVEMENT sel (PAY-IN)
      PAY-IN; read C;
    MOVEMENT alt (WITHDRAW)
      WITHDRAW; read C;
    MOVEMENT end
  CUSTOMER-BODY end
  TERMINATE;
CUSTOMER-1 end
```

Fig. 2B - Uso del linguaggio lineare per modellare il controllo all'interno di ogni processo.

Lo schema di Fig. 1 descrive un processo sequenziale (CUSTOMER), il cui comportamento è scomposto in una iterazione di una attività, a sua volta organizzata in tre sottoattività: la seconda è una iterazione di attività in conflitto.
Le figg. 2 invece modellano le interazioni del sistema da specificare con il mondo reale, nella forma grafica (fig. 2A) (molto vicina agli schemi data flow) e nella sua forma lineare (fig. 2B). La prima mette in rilievo la presenza di un processo "cliente reale" (CUSTOMER.0) e della sua immagine nel sistema (CUSTOMER.1) e del mezzo con cui queste due immagini comunicano, cioè del mezzo con cui i segnali dal mondo esterno "entrano" nel sistema per attivare azioni/procedure.
La seconda, invece, descrive l'intreccio tra attività di comunicazione (read C) rispetto alla struttura del processo sequenziale, alle sue attività interne.

Il metodo si sviluppa in una serie di passi, come mostrato in fig. 3.

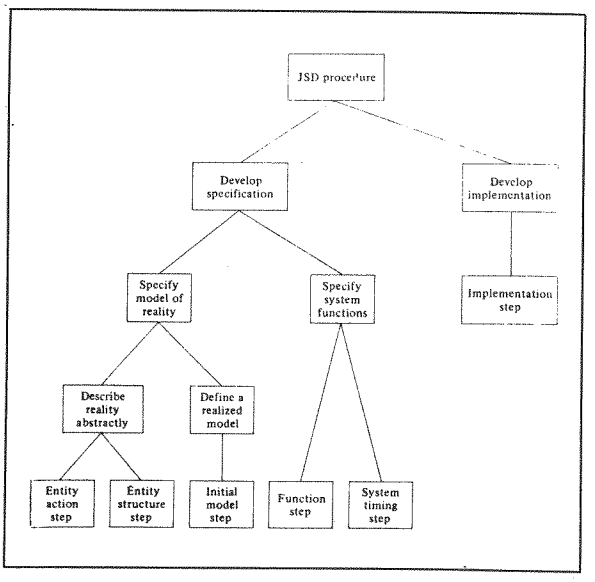
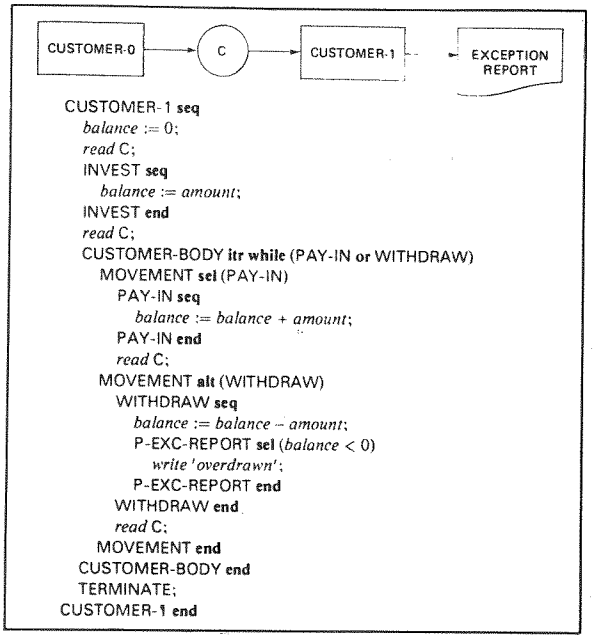


Fig. 3- La struttura del metodo JSD: la fase di specificazione, seguita da quella di realizzazione.

I linguaggi mostrati precedentemente vengono utilizzati per costruire il modello della realtà, al quale si possono aggiungere le prime specifiche relative alle funzioni che il sistema deve fornire all'ambiente ed i primi vincoli sui tempi con cui esse devono venire espletate.
Alcune volte tali funzioni sono semplicemente delle funzioni di output (come nell'esempio mostrato in fig. 4A). In questo caso allo schema data-flow, vengono semplicemente aggiunti dei blocchi di tipo "report", collegati con il processo virtuale (cioè del sistema da disegnare) che devono svolgere tale funzione.
La forma lineare descrive ancora una volta come questa funzione si inserisce nella attività del processo modellato.




```
CUSTOMER-1 seq
  balance := 0;
  read C;
  INVEST seq
    balance := amount;
  INVEST end
  read C;
  CUSTOMER-BODY itr while (PAY-IN or WITHDRAW)
    MOVEMENT sel (PAY-IN)
      PAY-IN seq
        balance := balance + amount;
      PAY-IN end
      read C;
    MOVEMENT alt (WITHDRAW)
      WITHDRAW seq
        balance := balance - amount;
        P-EXC-REPORT sel (balance < 0)
          write 'overdrawn';
        P-EXC-REPORT end
      WITHDRAW end
      read C;
    MOVEMENT end
  CUSTOMER-BODY end
  TERMINATE;
CUSTOMER-1 end
```

Fig. 4A - Inserimento delle funzioni verso l'ambiente. Il caso di un semplice incremento del modello precedente.

Altre volte richiedono l'introduzione di ulteriori processi e delle relative interazioni con i precedenti. In questo caso lo schema data-flow viene arricchito di nuovi processi (ad es. ENQUIRY FN Fig. 4B) con i relativi mezzi di comunicazione (ad. es. lo state vector CV) e le eventuali funzioni verso l'ambiente (ad es. ENQUIRY REPORT). Anche la descrizione lineare deve essere arricchita di un nuovo processo, con la sua struttura sequenziale e le azioni di comunicazione e di "report" (si veda sempre Fig. 4B).
I citati linguaggi però non modellano le specifiche sui tempi di attuazione.

A questo punto può partire la fase di implementazione. In essa si devono tenere conto i vincoli sulle risorse, legati alle caratteristiche del sistema di calcolo su cui il sistema finale dovrà venire eseguito.

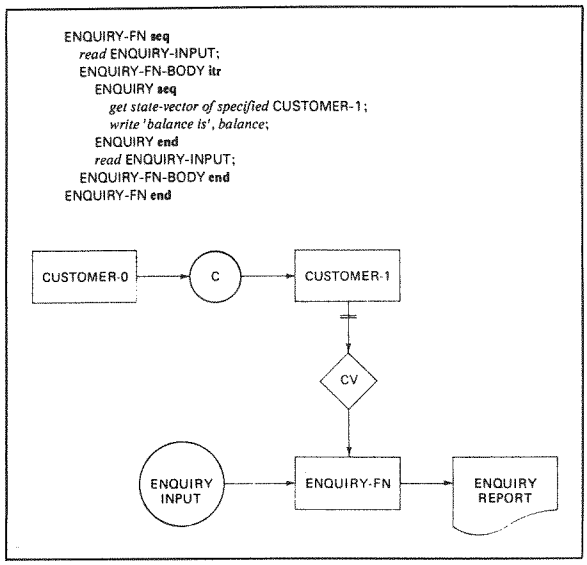


Fig. 4B - ... e il caso in cui un nuovo processo deve essere inserito, con la sua struttura di comunicazione.

In generale tali vincoli impongono di adeguare il precedente modello, che prevede tanti processi quanti sono gli esecutori logici, al numero di esecutori effettivi (il caso più sfavorevole è una singola macchina su cui lavora un singolo task). Questa operazione in genere implica l'introduzione di un processo scheduler che gestisce la distribuzione delle risorse tra i processi, che con opportune inversioni (mutuate dal già citato metodo JSP) diventano delle procedure gestite dallo scheduler. In fig. 5 è mostrato lo schema di flusso dei dati in una implementazione monotask/monomacchina dell'esempio presentato precedentemente. Il processo scheduler è il cuore del sistema, in quanto governa l'acquisizione dei dati da e verso l'esterno e gestisce l'attivazione dei processi, che secondo la classica filosofia della "program inversion" diventano procedure al servizio dello scheduler stesso. (I diversi tipi di collegamento tra scheduler e processi/procedure denotano diversi tipi di inversione).

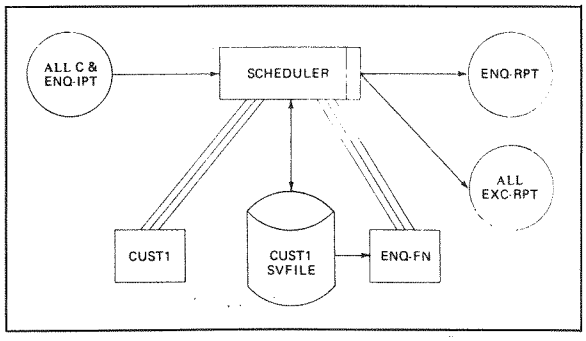


Fig. 5 - Struttura del sistema da realizzare - L'ipotesi di un solo esecutore impone la presenza di un processo scheduler, che governa l'esecuzione dell'intero sistema.

Ma anche lo scheduler è un processo, quindi è necessario modellarne la struttura rispetto alle funzioni che deve realizzare: una sua possibile struttura è mostrata in fig. 6. Una descrizione del controllo nel citato linguaggio lineare è il ponte verso l'uso di tradizionali linguaggi di programmazione.

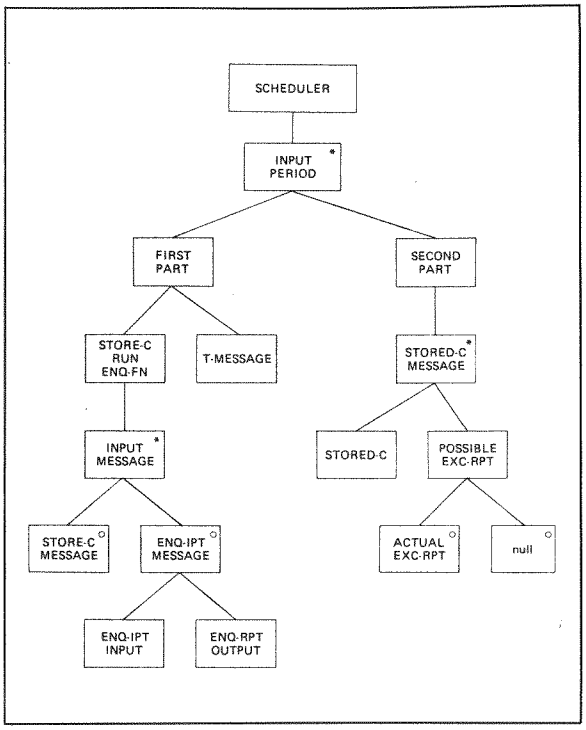


Fig. 6 - La struttura del processo scheduler, che determina la struttura del programma che lo realizza.

Può sorgere ora la domanda: quanto è strutturante JSD dal punto di vista del metodo di lavoro e della organizzazione che lo supporta?
Come afferma Jackson stesso, JSD tende a rompere le tradizionali gerarchie e i differenti pesi di potere e di retribuzione che le diverse figure di esperti assumono.
Non si tratta più di distinguere tra attività di programmazione o di analisi, ma tra specificazione ed implementazione: entrambe richiedono specifiche conoscenze tecniche, ma lo stesso livello di preparazione professionale. "... Semplificando un po', si può dire che le prime fasi di JSD sono gestite dall'utente con l'aiuto di uno specialista 'user oriented', mentre le ultime da una macchina con l'aiuto di uno specialista 'machine oriented',..." [3].

Proprio nella frase quotata sta l'essenza della proposta di JSD, le sue luci e le sue ombre. La proposta è accettabile per alcuni versi, ma discutibile per altri. Non si può non riconoscere che la critica agli altri approcci di modellazione del mondo reale sia profonda-

mente giustificata e catturi l'essenza di alcuni dei principali nodi del rapporto tra problema-modello-soluzione, che stanno alla base di molti fallimenti nella introduzione dell'informatica nelle strutture organizzative, dalle più semplici alle più complesse.

Ma ci si può chiedere se gli strumenti proposti in JSD sono adeguati a questo compito.
Prima di tutto la molteplicità dei linguaggi, non di per sè complessi, ma il cui uso può generare confusione nell'utente.
Secondariamente, l'assenza di qualunque criterio valutativo sui modelli, che consenta di confrontare diverse soluzioni: alcuni modelli non sono eseguibili e non consentono una descrizione delle interazioni tra processi, tipicamente quelli costruiti coi linguaggi grafici; altri consentono unicamente un approccio 'programming like', con elevata ambiguità sulle specifiche delle strutture dei dati, e con tutte le rigidità che tale approccio mostra nel modellare il mondo reale.

Da ultima, ma non meno importante, è l'assenza di una qualunque teoria all'interno della quale definire criteri di consistenza tra diversi modelli: questo è un punto veramente centrale, se si pensa che la forza dell'approccio dovrebbe essere nella facilità di automatizzare il processo implementativo e di facilitare, favorendo la modificabilità e la modularità dei modelli, la fase di specifica con l'utente.

Si ha l'impressione che la automazione sia pensata più come codifica di una classe di casi standard, piuttosto che come la costruzione di strumenti più generali che trovano le loro basi in una teoria.
La necessità di sintetizzare un approccio teorico con le esigenze applicative sembra ancora un enunciato, non un risultato.

Eppure la ricerca ha già messo a disposiozione strumenti linguistici e concettuali adeguati per rispondere alle esigenze di modellazione che stanno alla base di JSD.
Ad esempio, le reti di Petri [7] sono un felice connubio tra un linguaggio grafico di elevata fruibilità e una teoria matematica della concorrenza che permette di costruire, valutare e verificare modelli (anche in modo automatizzato) sulla base di parametri non puramente empirici.
In particolare l'approccio presentato in [8] sembra il più vicino alle premesse di JSD, in quanto non a caso ha nel già citato lavoro di Hoare le sue più remote origini [9].
In tale approccio da un lato sono specificati criteri di modellazione di sistemi organizzativi (legati al concetto di astrazione organizzativa), dall'altro criteri di valutazione dei modelli (legati ai concetti di autonomia e di rumore organizzativo). L'analisi del mondo reale (cioè della organizzazione che accoglie il sistema) è quindi governata da principi 'organization oriented'.

Dall'altro, il linguaggio utilizzato (le Reti di Automi Sovrapposti [10], una sottoclasse delle reti di Petri) consente di unificare in un unico linguaggio i tre aspetti di sequenzialità (arricchita dalla possibilità di descrivere il non determinismo), di trattamento della concorrenza (modellata come interazione sincrona tra processi) e di modellazione dei flussi di dati e della loro distribuzione tra i processi (utilizzando il frame delle reti ad alto livello, cioè la classe delle reti Predicati/Transizioni |7|

L'approccio proposto da Jackson con JSD è comunque un punto focale nel panorama delle metodologie per la costruzione dei sistemi informativi, decisamente innovativo per certi aspetti, ma bisognoso di profonde integrazioni per potersi presentare a tutti i diritti come un candidato per "Le nuove frontiere nello sviluppo dei sistemi informativi degli anni '80", come ambiziosamente si intitolava il già citato seminario di presentazione del metodo.

RIFERIMENTI

[1] P. Naur, B. Randell (eds), *"Software Engineering"*, Report on a Conference – NATO Science Commitee, 1969

[2] M. Jackson, *"Software Development as an Engineering Problem" Angewandte Informatik, 2/82, 1982*

[3] M. Jackson, *"System Development"*, Prentice Hall Series in Comp. Science, 1983

[4] S. Ceri (a cura di), *"Methodology and Tools for Data Base Design"*, North Holland, 1983

[5] C.A.R. Hoare, *"Communicating Sequential Processes"*, C. ACM 21,8,1978

[6] M. Jackson, *"Principles of Program Design"*, Academic Press, 1975

[7] G. Goos, J. Hartmanis (a cura di), *"Net Theory and Applications"* Lec. Not. Comp. Science 84, Springer, 1980

[8] F. De Cindio e altri, *"Conditions and Tools for an effective Negotiation during the organization/information System Design Process"*, in: C. Ciborra, I. Schneider, *"System Design for, with and by Users"*, North-Holland, 1983

[9] F. De Cindio e altri *"A Petri net model of CSP"*, Proc. CIL '81 Conference, Barcellona, 1981

[10] F. De Cindio e altri *"Superposed Automata Nets"*, Informatik – Fachberichte 52, Springer, 1982

OBJ, UN LINGUAGGIO DI PROGRAMMAZIONE BASATO SULLA LOGICA EQUAZIONALE

Giancarlo Mauri
Istituto di Cibernetica – Università di Milano

Sommario

OBJ è un linguaggio di programmazione ad altissimo livello di tipo funzionale; con caratteristiche di estremo interesse per l'utilizzo nell'ambito di una metodologia di programmazione avanzata, tra cui meccanismi di definizione di tipi di dati anche parametrici da parte dell'utente, facilitazioni per la programmazione gerarchica e modulare interattiva, strumenti di debugging interattivi molto potenti. In questo lavoro si presentano le caratteristiche concettuali fondamentali di OBJ, senza entrare nei dettagli delle teorie matematiche sulla base delle quali questo linguaggio è stato sviluppato.

Abstract

In this report, the fundamental aspects of the OBJ programming language, developed at SRI International by J. Goguen and others, are presented. OBJ is a very high level programming language based on the equational logic, hence functional in style, with user definable abstract data types and powerful tools for hierarchical development, modularization, interactive debugging of programs.

1. INTRODUZIONE

OBJ è un linguaggio nato nell'ambito delle ricerche sulla ingegneria del software e sulle metodologie di programmazione come linguaggio di specificazione che fornisse anche la possibilità di testare rapidamente i programmi specificati attraverso la esecuzione diretta di semplici tests ad alto livello. Come tale è stato proposto da J.A. Goguen ed implementato da J. Tardo presso la University of Southern California at Los Angeles, col nome OBJT [6,12,20].
Attualmente, è possibile vederlo come un linguaggio di programmazione ad altissimo livello, orientato ad architetture della quinta generazione, che sono certamente in grado di supportare implementazioni molto efficienti delle regole di riscrittura su cui si basa l'esecuzione di OBJ: si può insomma pensare ad una "OBJ machine" [10], esattamente come sono state costruite o sono in costruzione LISP machines e PROLOG machines.

Da un punto di vista molto generale, OBJ è analogo al LISP puro e, soprattutto, a PROLOG: è un linguaggio di programmazione "logico", nel senso che la descrizione ad alto livello di ciò che il programma dovrebbe fare è già il programma stesso, ed è quindi eseguibile direttamente. I vantaggi sono quelli ben noti dei linguaggi di questo tipo:

- separazione della struttura logica dei programmi dalla struttura di controllo, che semplifica la fase di disegno;
- coincidenza tra la logica del programma e la logica della prova del programma, che semplifica le dimostrazioni di correttezza;
- trasparenza dei programmi, che semplifica le eventuali fasi di modifica e manutenzione.

A differenza di PROLOG, che si basa sulla logica dei predicati non tipizzata, OBJ si basa una logica equazionale con tipizzazione forte, con caratteristiche di estremo interesse per l'uso nell'ambito di una metodologia di programmazione avanzata:

- possibilità di definizione di tipi di dati da parte dell'utente;
- possibilità di definizioni parametriche e di formazione di librerie di oggetti astratti, che incoraggia uno stile di programmazione basato sulla modularità e il riutilizzo dei moduli, massimizzando la chiarezza concettuale, la modificabilità e la facilità di debugging;
- possibilità di sviluppo gerarchico interattivo delle specifiche;
- possibilità di salvare qualunque stato del sistema OBJ e di usarlo come programma di utilità;
- possibilità di prototipizzazione rapida;
- disponibilità di strumenti per la definizione e il trattamento delle condizioni eccezionali;
- disponibilità di strumenti di programmazione e di debugging interattivi molto potenti.

Come si è detto, una prima implementazione di OBJ è stata realizzata da J. Tardo presso la UCLA [20].

La versione attuale, OBJ1, è stata sviluppata da D. Plaisted [10] presso lo SRI International.

Mentre le motivazioni per lo sviluppo di questo linguaggio vengono dalla ingegneria del software, gran parte del know-how che ne ha reso possibile il disegno è legato allo sviluppo di teorie matematiche piuttosto sofisticate sui tipi di dati astratti e sulle basi matematiche della programmazione che non è possibile trattare in questa sede. Il lavoro ha quindi come obiettivo principale quello di dare una idea della filosofia generale del linguaggio e delle sue caratteristiche concettuali essenziali, rispetto alle quali è inessenziale la presentazione e la piena comprensione dei dettagli e degli strumenti matematici sottostanti.

2. LA SCRITTURA DELLE SPECIFICHE

Nel corso degli anni '70 si è andata gradualmente precisando, e si è travasata altrettanto gradualmente nei linguaggi di programmazione, l'idea che un tipo di dati è costituito da due elementi inscindibili: i dati veri e propri, che possono essere raggruppati in insiemi diversi (chiameremo *sorte* questi insiemi di dati) e le operazioni che su questi dati si possono eseguire. Ad esempio, il tipo STACK è caratterizzato da due sorte, l'insieme dei dati elementari e l'insieme degli stacks, e dalle operazioni PUSH, che aggiunge un elemento ad uno stack, POP, che toglie un elemento da uno stack, e TOP, che legge l'elemento in cima ad uno stack. È inoltre diventato chiaro che la rappresentazione di un tipo di dati è cosa diversa dal tipo di dati: questo infatti può essere rappresentato in più modi distinti. Poichè lo stesso insieme di dati con operazioni diverse è un tipo di dati diverso, è anche necessario "proteggere" i dati dalla possibilità di manipolarli attraverso operazioni diverse da quelle del tipo cui appartengono, ad esempio operazioni che manipolano la rappresentazione dei dati. È questa la essenza della astrazione sui dati, che sembra definitivamente acquisita nei linguaggi di programmazione più recenti (CLU, Alphard, ADA,...) e che in OBJ trova espressione in una forma molto avanzata.

Un programma OBJ è costituito da un insieme di oggetti astratti modulari, che possono essere tipi o moduli, specificabili dall'utente. Per specificare questi oggetti, OBJ offre una serie di facilitazioni, tra cui in particolare:

- possibilità di creare nuove specifiche arricchendo od estendendo oggetti specificati in precedenza con nuove operazioni e/o sorte;
- possibilità di scrivere specifiche parametriche;
- strumenti (semi) automatici per il controllo statico della consistenza sintattica e semantica delle specifiche;
- possibilità di manipolare interattivamente, editare, salvare in files le specifiche; inoltre, nell'uso interattivo, l'utente riceve un feedback immediato attraverso messaggi di errore informativi.

2.1. La sintassi delle specifiche

La specificazione di un oggetto ha la seguente forma sintattica:

OBJ (nome dell'oggetto) / (lista di nomi di oggetti)
SORTS (lista delle sorte)
OK-OPS (lista di operazioni)
ERROR-OPS (lista di operazioni)
VARS (lista di identificatori)
OK-EQNS (lista di equazioni)
ERROR–EQNS (lista di equazioni)
JBO

OBJ e JBO indicano l'inizio e la fine della definizione. Discutiamo ora il significato degli altri elementi della definizione.

2.1.1. La gerarchia degli oggetti

Dopo lo slash (/) vengono elencati gli oggetti definiti in precedenza che vengono utilizzati nella definizione; tra gli oggetti esiste quindi una relazione gerarchica, rappresentabile attraverso un grafo aciclico con radice. La radice è costituita dall'oggetto TRUTH, incorporato nel sistema e gerarchicamente sottostante ad ogni altro oggetto, che contiene i valori di verità T e F (ma non le operazioni su di essi) ed è sempre disponibile senza necessità di richiamo da parte dell'utente.
Nel sistema sono incorporati anche gli oggetti BOOL (algebra booleana con due elementi), NAT (numeri naturali), INT (interi) e ID (identificatori), che però devono essere richiamati esplicitamente e possono anche essere ridefiniti dall'utente.

Questa organizzazione gerarchica consente di raggiungere un alto livello di modularità in modo naturale.
Interessante è anche la possibilità di coesistenza di sottogerarchie rappresentanti elaborazioni mutuamente inconsistenti, il che consente di rendere disponibile più di un modo per fare la stessa cosa, rinviando la scelta su quale utilizzare effettivamente ad un momento successivo dello sviluppo del programma, quando si disporrà di ulteriori informazioni per effettuare la scelta.

Si osservi che l'obbligo di utilizzare oggetti già definiti per definire nuovi oggetti rappresenta un vincolo piuttosto forte, che impone un rigido meccanismo bottom up nella costruzione delle specifiche.

2.1.2. Sorte e operazioni

Ogni oggetto definisce una o più nuove sorte di dati, insieme con le funzioni di base che servono per creare, selezionare, manipolare tali dati. Le operazioni vengono listate suddivise in due sezioni, operazioni ok e operazioni di errore (su ciò si tornerà in seguito) e per ognuna di esse sono indicate la arità (cioè il dominio e il codominio) e la forma (cioè la collocazione degli argomenti rispetto al simbolo di operazione). Poichè uno degli obiettivi è la naturalezza nella scrittura delle specifiche, la sintassi della forma delle operazioni è molto flessibile, e consente di utilizzare, oltre alla normale forma parentesizzata in cui si rappresentano le funzioni, f (x_1, x_2,...), anche altri tipi di notazione: infissa, prefissa, postfissa o mista.

ESEMPI:

POP – : STACK —› STACK prefissa

–! : INT —› INT postfissa

–AND– : BOOL BOOL —› BOOL infissa

IF-THEN-ELSE-FI : BOOL INT INT —› INT mista

(il segnaposto – indica la posizione degli argomenti)

Il sistema provvede inoltre automaticamente a corredare ogni sorta definita di tre operatori: il predicato di uguaglianza – = – , il predicato di errore ERR–, l'operatore condizionale IF – THEN – ELSE – FI. Infine, nonostante la tipizzazione forte su cui si basa, OBJ consente l'utilizzo di "coercions", che definiscono una sorta come sottoinsieme di un'altra, estendendo ad essa le relative operazioni.

ESEMPIO

– : INT —› COMPLIN

specifica gli interi come sottoinsieme degli interi complessi, dice cioè che ogni numero intero x (l'argomento, da sostituire al segnaposto –) può essere pensato come una abbreviazione per il numero complesso x+i.O.

Le ambiguità che ne conseguono (poichè un simbolo di operazione può essere inteso come relativo a più sorte diverse) sono tollerate dal parser; viene però controllata la assenza di cicli del tipo:

– : A —›B ; – : B —› A

2.1.3. Le equazioni

Le equazioni, rappresentate da uguaglianze tra due espressioni contententi le variabili dichiarate nella sezione VAR, oltre ai nomi di operazioni dell'oggetto, definiscono le proprietà delle operazioni. Anche in questo caso abbiamo equazioni ok, che valgono tra espressioni che non sono errori, ed equazioni che definiscono le condizioni di errore ed il modo in cui trattarle. Per semplificare la scrittura di specifiche, il sistema mette a disposizione alcuni attributi, equivalenti ad una o più equazioni, che possono essere assegnati alle operazioni. ASSOCIATIVE, COMMUTATIVE, IDEMPOTENT sono applicabili a operatori binari, cui assegnano le proprietà di associatività, commutatività, idempotenza (ricordiamo che una operazione binaria è idempotente se per ogni x vale x.x =x); EQUALITY, applicabile a operatori binari booleani, equivale alle tre proprietà definitorie dell'ugualianza; HIDDEN rende l'operatore non disponibile al di fuori dell'oggetto in cui è dichiarato.

L'occultamento di informazione così ottenuto ha un profondo significato metodologico, discusso ad esempio da Parnas [18], e consente inoltre di aumentare la potenza delle specifiche equazionali, allargando così la classe dei tipi di dati definibili equazionalmente [21].

2.2. Il meccanismo di parametrizzazione

È possibile definire oggetti parametrici, in cui alcune sorte sono fornite come parametri al momento della specifica e possono essere istanziate in seguito. Il costrutto IMAGE (abbreviato in IM) consente di ottenere istanze effettive degli oggetti parametrici, assegnando valori precisi ai parametri. Anche l'uso di moduli astratti parametrizzati è un potente strumento di strutturazione delle specifiche, particolarmente utile per la costruzione di librerie di programmi riutilizzabili.

L'esempio 1 definisce una lista i cui elementi sono indicati in forma parametrica: in pratica, un lista viene definita come un contenitore che può essere riempito con oggetti di qualunque tipo.

Il significato della specificazione è chiaro. Abbiamo anzitutto due sorte, LIST ed ELEMENT. LIST è l'insieme delle liste costruite con elementi appartenenti all'insieme ELEMENT, che è un parametro, cui si potrà assegnare un valore in seguito: come elementi di una lista si potranno prendere interi, identificatori, liste, arrays e così via.

```
ESEMPIO 1

  OBJ LIST
  SORTS LIST ELEMENT
  OK-OPS
    NIL : —› LIST
    – : ELEMENT —› LIST
    –;– : LIST LIST —› LIST (ASSOCIATIVE)

    FIRST : LIST —› ELEMENT
    REST : LIST —› LIST
  ERR-OPS
    NO-FIRST : —› ELEMENT
    NO REST : —› LIST
  VARS
    L : LIST
    E : ELEMENT
  OK-EQNS
    (NIL ; L = L)
    (L ; NIL = L)
    (FIRST (E ; L) = E)
    (REST (E ; L) = L)
    (FIRST (E) = E)
    (REST (E) = NIL)
  ERR-EQNS
    (FIRST (NIL) = NO-FIRST)
    (REST (NIL) = NO-REST)

  JBO
```

Inoltre, sulle liste è possibile eseguire una operazione binaria ";" che concatena due liste generando una nuova lista, una operazione FIRST che estrae il primo elemento di una lista ed una operazione REST che applicata ad una lista la priva del primo elemento; inoltre, ogni elemento può essere interpretato come una lista formata da un solo elemento. Anche le equazioni sono di immediata comprensione: la concatenazione di una lista L con la lista vuota a destra o a sinistra lascia invariata L; il primo elemento ed il resto della lista E; L ottenuta aggiungendo l'elemento E alla lista L sono rispettivamente E ed L, mentre il primo elemento ed il resto della lista costituita dall'unico elemento E sono rispettivamente E e la lista vuota NIL: infine, il tentativo di estrarre il primo elemento od il resto dalla lista vuota è dichiarato errore. Si noti anche l'uso dell'attributo ASSOCIATIVE per l'operazione di concatenazione di liste, che consente di non scrivere esplicitamente l'equazione di associatività.

Per definire una istanza specifica con il costrutto IM occorre specificare: un nome per il nuovo oggetto, nuovi nomi di sorta in corrispodenza dei nomi parametrici, ed eventualmente nuove equazioni, con la seguente sintassi:

```
IM (‹nome› ⇒ ‹nome›) / ‹lista di nomi›
SORTS (‹sorta› ⇒ ‹sorta›)
OPS (‹forma› ⇒ ‹forma›)
MI
```

ID è in sostanza un parametro attuale che viene assegnato come valore al parametro formale ELEMENT. La freccia ⇒ indica che ELEMENT ha come immagimne ID.

Nell'esempio seguente, partendo dall'oggetto parametrico LIST ed utilizzando l'oggetto built-in ID, si specifica l'oggetto attuale ID-LIST (lista di identificatori):

```
IM (LIST ⇒ ID-LIST) / ID
SORTS (LIST ⇒ ID-LIST)
       (ELEMENT ⇒ ID)
MI
```

3. LA SEMANTICA MATEMATICA DI OBJ

Come si è accennato nella introduzione, uno degli aspetti più importanti di OBJ è il fatto che esso è stato disegnato tenendo conto anche della necessità di fornire solide basi matematiche in particolare per garantire la correttezza dei programmi. Uno dei più importanti riferimenti concettuali a questo proposito è costituito dalle ricerche sulla semantica dei programmi che hanno portato a individuare due atteggiamenti distinti, e in parte contrapposti, l'atteggiamento "denotazionale", di tipo spiccatamente matematico, e l'atteggiamento "operazionale", che tiene maggiormente conto degli aspetti di esecuzione dei programmi. Ambedue gli atteggiamenti, che non possiamo discutere più estesamente, presentano aspetti positivi.
Proprio per questo, OBJ vuole essere ben fondato sia dal punto di vista denotazionale che da quello operazionale.

La disponibilità di una semantica denotazionale per un linguaggio di programmazione consente di assegnare un significato preciso ai programmi in un modo concettualmente chiaro e semplice, e di usare le tecniche dimostrative del sistema logico sottostante per provare le proprietà dei programmi.

OBJ si basa su una semantica denotazionale di tipo algebrico: la denotazione di un programma OBJ è una struttura algebrica eterogenea, cioè una collezione di insiemi sui quali è possibile eseguire alcune operazioni, che sono inseparabili dai dati.

Le basi teoriche della specifica di tipi di dati astratti e di programmi su di essi attraverso strutture algebriche sono state studiate da Goguen, Thatcher, Wa-

gner e Wright [13,21], Wand [22], Bertoni, Mauri e Miglioli [1], Broy, Dosch, Pepper, Partsch e Wirsing [2] e altri.

Senza entrare qui nei dettagli matematici della questione, possiamo limitarci a ricordare che i problemi principali riguardano la formulazione delle proprietà richieste per le operazioni e la esistenza ed unicità della struttura specificata da un programma OBJ. Poichè in generale esistono più strutture associate ad una specificazione, in OBJ si assume come semantica la struttura iniziale, che è unica a meno di isomorfismi (se esiste) e rappresenta in un certo senso la struttura che gode esattamente delle proprietà richieste nella specificazione e di nessun'altra. Per quanto riguarda l'esistenza di questa struttura, si dimostra che è garantita nel caso in cui le proprietà sono espresse attraverso equazioni o equazioni condizionali, del tipo:

$$GCD(I,J) = GCD(I-J,J) \text{ IF } I > J$$

ove GCD(I,J) indica il massimo comune divisore tra I e J.

Non tutti i costrutti di OBJ hanno però una semantica matematica completa. In particolare, per dare una semantica soddisfacente per il costrutto IMAGE è necessario superare il livello dell'algebra universale, utilizzando invece la teoria delle categorie e le teorie algebriche, seguendo ad esempio le proposte di Burstall e Goguen per la semantica di CLEAR, un linguaggio di specificazione con basi matematiche più profonde di OBJ [3]. Inoltre, anche la teoria delle algebre con errori (vedi par. 6) non ha ancora raggiunto una sistemazione completa e soddisfacente.

4. LA SEMANTICA OPERAZIONALE

Una semplice semantica operazionale per le strutture di dati definite in termini equazionali è ottenibile interpretando le equazioni come regole di riscrittura.

Ad esempio, l'equazione

$$POP\ (PUSH\ (I,S)) = S$$

può essere utilizzata per effettuare la riscrittura:

$$TOP\ (POP\ (PUSH\ (2,\ PUSH\ (1,\ EMPTY))) \Rightarrow TOP\ (PUSH\ (1,EMPTY))$$

Questa semantica operazionale è coerente con la semantica denotazionale se per il sistema di riscrittura associato all'insieme di equazioni che definiscono l'oggetto valgono le seguenti due proprietà (per una definizione rigorosa di sistema di riscrittura associato ad un insieme di equazioni e per la dimostrazione formale dei risultati seguenti si può vedere [20]):

- non vi sono espressioni che generano sequenze di riscrittura infinite (proprietà di terminazione finita)
- tutte le sequenze di riscrittura sulla stessa espressione danno lo stesso risultato (proprietà di Church-Rosser).

Anche questa semantica operazionale è giustificata da un lavoro teorico non indifferente, ad opera di Huet [14], Wand [22], O'Donnel [17], Goguen [5].

Per quanto riguarda l'implementazione, essa si basa sull'algoritmo di Knuth-Bendix [15] per determinare la confluenza.
La terminazione costituisce un problema più serio, e non ancora completamente risolto in modo soddisfacente. Infatti il test di Lankford, che viene utilizzato, è un test parziale (cosa ovvia, data la indecidibilità della terminazione in generale) che potrebbe forse essere migliorato attraverso l'uso di euristiche adeguate.

Un caso molto frequente di non terminazione è dovuto alla presenza di equazioni di commutatività:

$$X.Y = Y.X$$

che possono dare origine a sequenze di riscrittura del tipo:

$$A.B \Rightarrow B.A \Rightarrow A.B \Rightarrow \ldots$$

Per riconoscere queste situazioni si può utilizzare il comando RUM (vedi par. 5).

5. IL TEST DELLE SPECIFICHE

La valutazione di espressioni di prova consente di debuggare le specifiche, controllandone la completezza e la consistenza. L'espressione da valutare viene fornita racchiusa tra le parole chiave RUN e NUR. Il sistema di tipizzazione forte su cui si basa OBJ consente il controllo statico della struttura dell'espressione; si effettua anzitutto il parsing dell'espressione; se non è univoco, il parser instaura un dialogo interattivo con l'utente per eliminare le ambiguità, fornendo informazioni sulle difficoltà che incontra e permettendo di effettuare correzioni.

Ad esempio, dopo aver specificato il GCD come segue:

```
OBJ GCD/INT          Nome oggetto/ sorte richieste
OK-OPS                    Operazioni lecite
    GCD : INT, INT —> INT con le relative arità
ERR-OPS                   Operatori di errore
    NEG-ARG : —> INT      (argomenti negativi)
VARS                 Dichiarazione di variabili
    I J : INT
```

```
OK-EQNS                          Lista di equazioni
    (GCD(I,I)=I)
    (GCD(I,O)=I)
    (GCD(I,O)=I)
    (GCD(I,J))=GCD(I–J,J)) IF I›J)
    (GCD(I,J))=GCD(I,J–I)) IF I‹J)
ERR-EQNS                         Equazioni di errore
    (GCD(I,J)=NEG-ARG IF I‹J + J‹O)
```

JBO

la richiesta di valutazione:

RUN GCD (1308,156) NUR

dà come risultato:

AS INT : 12

attraverso la sequenza di riscrittura:

GCD (1308,156) ⇒ GCD (1152,156) ⇒ GCD (996,156) ⇒ GCD (216,156) ⇒ GCD (60,156) ⇒ GCD (60,36) ⇒ GCD (24,36) ⇒ GCD (24,12) ⇒ GCD (12,12) ⇒ 12

Il comando RUM (RUn with Memory) ricorda tutti i termini ottenuti nel corso della riscrittura, e si arresta quando incontra un termine ripetuto, consentendo così di evitare loops. È ovviamente molto costoso in termini sia di tempo che di spazio.

Un'ultima opzione interessante è la possibilità di avere la traccia delle sequenze di valutazione. In questo caso vengono man mano stampate le sottoespressioni valutate, indentate per indicare il livello di annidamento. Inoltre, vengono evidenziati l'equazione applicata ed il risultato di ogni passo di riscrittura.

6. IL TRATTAMENTO DELLE CONDIZIONI ECCEZIONALI

Le operazioni totali dell'algebra universale classica sono poco adeguate a modellare le operazioni tipicamente occorrenti nella specifica dei programmi, che sono spesso parziali. Per risolvere questo problema, Goguen ha sviluppato una teoria delle algebre con errori [6], che non ha tuttavia la completezza e l'eleganza della teoria delle algebre totali e presenta ancora molte lacune.

Questa teoria fornisce la base per il riconoscimento e la segnalazione delle condizioni eccezionali in OBJ. Le condizioni eccezionali (tentativo di applicare operazioni ad argomenti di tipo scorretto o per cui non sono definite, overflow, underflow etc.) sono definite e trattate attraverso equazioni di errore, in cui l'informazione sull'errore è codificata attraverso speciali "operatori di errore", che possono produrre termini che sono messaggi di errore complessi. In particolare, devono essere specificati nel modo più dettagliato possibile sia il tipo di errore, sia la posizione (sottoespressione) in cui compare. Ad esempio, il tentativo di applicare GCD a numeri negativi, per cui non è definito, come in:

RUN GCD(22,–11) NUR

oppure in

RUN 1+GCD(22,GCD(2191,1533)–18)) NUR

darebbe origine rispettivamente alle seguenti risposte:

AS INT : »ERROR» NEG-ARG

AS INT : »ERROR» 1+NEG-ARG

Sono stati fatti anche tentativi di includere il trattamento ed il recovery delle situazioni eccezionali, ma su questo punto i risultati sono ancora scarsi.

7. APPLICAZIONI E POSSIBILI SVILUPPI

OBJ è stato utilizzato sperimentalmente per la specificazione di alcuni sistemi; un primo esempio è costituito dalla specificazione di linguaggi di programmazione. In [9], ad esempio, è riportata la specifica di un semplicissimo linguaggio per le funzioni ricorsive sugli interi, FUN.

Il linguaggio FUN consente di definire qualsiasi funzione ricorsiva sugli interi.
La specificazione del linguaggio utilizza due oggetti parametrici, ARRAY e LIST, che si suppongono disponibili in libreria. L'oggetto parametrico LIST viene istanziato in tre modi diversi, dando origine ai tre oggetti IDENT (liste di identificatori), IDENTL (liste di liste di identificatori) e INTL (liste di interi), mentre ARRAY viene istanziato nell'oggetto ENV (che rappresenta l'environment), i cui elementi sono arrays di interi indicati da liste di identificatori.
Vengono quindi specificati i tre oggetti principali: EXP, che specifica le espressioni, suddivise nelle due sorte INTEXP (espressioni intere) e BOOLEXP (espressioni booleane), STMT, corrispondente ai comandi, con le operazioni di concatenazione, assegnamento, while do ed esecuzione di un comando in un dato environment, ed infine l'oggetto principale, basato su tutti i precedenti, i cui elementi sono i programmi FUN.
La specificazione completa, tratta da [9], con alcuni programmi di test è fornita in Figura 1.

Goguen e Parsaye-Ghomi [11] hanno fornito anche una specificazione di un linguaggio più complesso, MODEST. In ambedue i casi, la esecuzione della specifica consente di generare un prototipo di interprete.

Altri prototipi sviluppati riguardano sistemi di accesso a basi di dati, message systems e sistemi operativi sicuri [9,10].

Un particolare interesse sembrano avere le applicazioni alla intelligenza artificiale, in particolare per la semplicità con cui si possono specificare in OBJ algoritmi di ricerca in backtracking, algoritmi di pattern matching e "demoni", cioè pezzi di codice che vengono richiamati quando si verificano improvvisamente particolari condizioni.
Inoltre la sematica iniziale di OBJ è legata a ricerche recenti di Reiter [19], Clark [4] e McCarthy [16] sulla "assunzione del mondo chiuso" in intelligenza artificiale, sulla assunzione cioè che l'informazione effettivamente disponibile per un certo problema comprende tutte le informazioni necessarie per risolverlo. Il "mondo chiuso" è identificabile con modello iniziale della teoria generata dalla formulazione del problema, purchè tale teoria sia equazionale.

Infine, OBJ si presta particolarmente bene per la programmazione di robot, poichè in questo caso si hanno grosse esigenze di trattamento di condizioni eccezionali e di parallelismo.

```
*** This specification of the fun language uses parameterized objects list and stack from the OBJ Library ***

IM (LIST ⇒ IDENT)/ BOOL INT
SORTS (LIST ⇒ IDENT)
       (ELM⇒ ID)
IM (ARRAY ⇒ ENV)/ IDENT INT BOOL
SORTS (INDEX ⇒ IDENT)
       (ELM ⇒ INT)
       (ARRAY⇒ ENV)
MI

IM (LIST⇒ IDENTL)/ IDENT INT BOOL
SORTS (ELM ⇒ IDENT)
       (LIST ⇒ IDENTL)
OPS    (— : LIST LIST → LIST ⇒ –;–)
MI

IM (LIST ⇒ INTL)/ INT BOOL
SORTS (ELM ⇒ INT)
       (LIST ⇒ INTL)
OPS    (— : LIST LIST → LIST ⇒ –;–)
MI

OBJ EXP / ENV INT BOOL IDENT
SORTS INTEXP BOOLEXP
OK-OPS
    – : INT → INTEXP
    – : IDENT → INTEXP
    – : BOOL → BOOLEXP
    –AND– : BOOLEXP BOOLEXP → BOOLEXP
    –OR– : BOOLEXP BOOLEXP → BOOLEXP
    NOT– : BOOLEXP → BOOLEXP
    – ≤ –: INTEXP INTEXP → BOOLEXP
    –=– : INTEXP INTEXP → BOOLEXP
    IF–THEN–ELSE–FI : BOOLEXP INTEXP INTEXP → INTEXP
    –+– : INTEXP INTEXP → INTEXP
    – =–: INTEXP INTEXP → INTEXP
    –*– : INTEXP INTEXP → INTEXP
    ⊢ . [–] : INTEXP ENV → INT
    ⊢ . [–] : BOOLEXP ENV → BOOL
ERR-OPS
    NIL-ID : → INTEXP
VARS
    INT : INT
    BOOL : BOOL
    ID : IDENT
    EX EX' : INTEXP
    BX BX' : BOOLEXP
    ENV : ENV
OK-EQNS
    ((I INT ! . [ ENV ]) = INT)
    (I ID ! . [ ENV ] = ENV [ ID ])
    (I (EX + EX')! . [ ENV ] = (I EX ! . [ ENV ])+(I EX' ! . [ ENV ]))
    (I (EX – EX')! . [ ENV ] = (I EX ! . [ ENV ])–(I EX' ! . [ ENV ]))
    (I (EX * EX')! . [ ENV ] = (I EX ! . [ ENV ])*(I EX' ! . [ ENV ]))
    (I BOOL ! . [ ENV ] = BOOL)
    (I EX ‹ EX' ! . [ ENV ] = ! . [ ENV ] ‹ I EX' ! . [ ENV ])
    (I EX = EX' ! . [ ENV ] = I EX ! . [ ENV ] ≐= I EX' ! . [ ENV ])
    ((I BX AND BX' ! . [ ENV ])=(I BX ! . [ ENV ])AND(I BX' ! . [ ENV ]))
    ((I (BX OR BX') ! . [ ENV ])=(I BX ! . [ ENV ])OR(I BX' ! . [ ENV ]))
    ((I (NOT BX)' ! . [ ENV ])= NOT (I BX ! . [ ENV ]))
    (I IF BX THEN EX ELSE EX' FI ! . [ ENV ] =
      IF(I BX ! . [ ENV ]) THEN(I EX ! . [ ENV ])ELSE(I EX' !
      . [ ENV ]) FI)
ERR-EQNS

    - (NIL = NIL-ID)

JBO

OBJ STMT / EXP
SORTS STMT
OK-OPS
    –;– : STMT STMT → STMT (ASSOC)
    –:=– : IDENT INTEXP → STMT
    WHILE–DO–OD : BOOLEXP STMT → STMT
    ⊢ . [–] : STMT ENV → ENV
VARS
    S S' : STMT
    ID : IDENT
    EX : INTEXP
    BX : BOOLEXT
    ENV : ENV

OK-EQNS
    (I ID := EX ! . [ ENV ] = PUT(ID,I EX ! . [ ENV ],ENV))
    ((S ; S')! . [ ENV ] = I S' ! . [(I S ! . [(I S ! . [ ENV ])])
    ((WHILE BX DO S OD)! . [ ENV ] = IF(I BX ! . [ ENV ])THEN(I WHILE BX DO S OD ! . [(I S ! . [ ENV
    ])])ELSE ENV FI)

JBO

OBJ FUN / STMT IDENTL INTL
SORTS INIT 1 INIT FUN
OK-OPS
    NILINIT : → INIT
    –INITIALLY : IDENT INTEXP → INIT 1
    – : INIT 1 → INIT
    –;– : INIT INIT → INIT (ASSOC)
    FUN–––––VARS–BODY–NUF : IDENT IDENTL INIT STMT → FUN
    ⊢ . ⊢ . [–] : IDENTL INTL ENV → ENV
    ⊢ . [–] : INIT ENV → ENV
    ⊢ . [–] . ‹ › : FUN ENV INTL → ENV
    ⊢ . ⟿ : FUN INTL → INT
ERR-OPS
    WRONG/ARGS : → ENV
VARS
    ID ID' : IDENT
    IDL : IDENTL
    INT : INT
    INTL : INTL
    EX : INTEXP
    INIT1 : INIT1
    INIT : INIT
    STMT : STMT
    ENV : ENV
OK-EQNS
    ((I ID INITIALLY EX ! . [ ENV ])=(I ID := EX ! . [ ENV ]))
    (I INIT1 ; INIT ! . [ ENV ] = I INIT ! . [ I INIT1 ! . [ ENV ] ])
    ((I ID ! . I INT ! . [ ENV ])=(I ID := INT ! . . ] ENV ]))
    ((I(ID ; IDL)! . I (INT ; INTL)! . [ENV ])=(I IDL ! . I INTL ! . [(I ID ! . I INT ! [ ENV ])]))
    (I FUN ID ‹ IDL ›VARS NILINIT BODY STMT NUF ! . [ ENV ],‹ INTL› = I STMT ! . [ ENV ])
    (I FUN ID ‹ IDL ›VARS INIT BODY STMT NUF ! . [ ENV ],‹ INTL › = I STMT ! . [ I IDL ! .I INTL ! .
      . [ ENV ] ] ! )
    (I FUN ID ‹ IDL ›VARS INIT BODY STMT NUF ! . ‹ INTL › = I FUN ID ‹ IDL ›VARS INIT BODY STMT
    NUF ! . [ NILARR [ . ‹ INTIL › [ ID ]

I RR-EQNS
    (I IDL ! . I INTL ! . [ ENV ] = WRONG/ARGS IF NOT (LENGTH IDL == LENGTH INTL) OR INTL =
    = NIL)

    JBO

JN FUNP NI

*** TEST PROGRAMS FOR THE FUN DEFINITION ***

*** POW(N ; M) GIVES THE NTH POWER OF M FOR N POSITIVE OR O ***
RUN I FUN 'POW ‹ 'N ; 'M ›VARS 'POW INITIALLY 1 BODY WHILE O ‹ 'N DO 'POW := 'POW * 'M ; 'N
    := 'N – 1 OD NUF . ! . ‹ 4 : 2 › NUR

AS INT: 16

*** FACTORIAL OF N ***
RUN I FUN 'FAC ‹ 'N ›VARS 'FACT INITIALLY 1 ; 'INITIALLY O BODY WHILE 'I ‹ 'N DO 'I := 'I ; 'FAC
    : = 'I * 'FAC OD NUF ! . ‹ 5 › NUR

AS INT: 120

*** MAX GIVES THE MAXIMUM OF A LIST OF THREE NUMBERS ***
RUN I FUN 'MAX ‹ 'A ; 'B ; 'C ›VARS 'N INITIALLY 2 BODY 'MAX := 'A ;(WHILE O ‹ 'N DO('N := 'N
– 1 ; 'X := 'B ; 'B : = 'C ;('MAX := IF 'X ‹ 'MAX THEN 'MAX EL SE 'X FI))OD)NUF ? . › 3 ; 123 ; 32 › NUR

AS INT: 123
```

Fig. 1

8. BIBLIOGRAFIA

[1] A. Bertoni, G. Mauri, P. Miglioli, *"On the power of model theory in specifying abstract data types and in capturing their recursiveness"*, Fundamenta Informaticae, VI.2, 1983, pp. 127-170

[2] M. Broy, W. Dosch, H. Partsch, P. Pepper, M. Wirsing, *"Existential quantifiers in abstract data types"*, Proc. 6th ICALP, Lect. Not. in Comp. Sci. 71, pp. 73-87, Springer, Berlin, 1979.

[3] R.M. Burstall, J.A. Goguen, *"The semantics of Clear, a specification language"*, Lect. Not. in Comp. Sci. 86, pp. 292-332, Springer, Berlin, 1980.

[4] K.L. Clark, Negation as failure, in *"Logic in data bases"* (H. Gallaire and J. Minker ed's), pp. 293-322, Plenum Press, 1978.

[5] J.A. Goguen, *"How to prove algebraic inductive hypotheses without induction"*, Lect. Not. in Comp. Sci. 87, pp. 356-373, Springer, 1980

[6] J.A. Goguen, *"Abstract errors for abstract data types"*, IFIP Working Conf. on Formal Description of Programming Concepts, North Holland, 1977

[7] J.A. Goguen, *"Merged views, closed worlds and ordered sorts: some novel database features in OBJ"*, Tech: Rep., Comp. Sci. Lab., SRI International, 1983

[8] J.A. Goguen, J. Meseguer, *"Toward OBJ1, a study in executable algebraic formal specifications"*, Tech. Rep., Comp. Sci. Lab., SRI Int., 1981

[9] J.A. Goguen, J. Meseguer, *"Rapid Prototyping in the OBJ Executable Specification Language"*, Proc. Rapid Prototyping Workshop, 1982

[10] J.A. Goguen, J. Meseguer, D. Plaisted, *"Programming with parameterized abstract object in OBJ"*, Tech. Rep., Comp. Sci. Lab., SRI International, 1983

[11] J.A. Goguen, K. Parsaye-Ghomi, *"Algebraic denotational semantics using parameterized abstract modules"*, Lect. Not. in Comp. Sci. 107, pp. 292-309, Springer, Berlin, 1981

[12] J.A. Goguen, J. Tardo, *"An introduction to OBJ, a language for writing and testing software specifications"*, Proc. IEEE Conf. on Specification of Reliable Software, pp. 107-189, 1979

[13] J.A. Goguen, J.W. Thatcher, E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, in *"Current trends in programming methodology"* (R.T. Yeh ed.) vol. 4, Prentice Hall, Englewood Cliffs, 1978

[14] G. Huet, *"Confluent reductions"*, Proc. 18th IEEE Symp. on Fundations of Computer Science, 1977

[15] D. Knuth, P. Bendix, Simple world problems in Universal Algebra, in *"Computational problems in abstract algebra"* (J. Leech ed.), Pergamon Press, 1970

[16] J. Mc Carthy, Circumscription, *"A form of non monotonic reasoning"*, Artificial Intelligence 13, pp. 27-39, 1980

[17] M.J. O'Donnell, *"Computing in Systems described by equations"*, Lect. Not. in Comp. Sci. 58, Springer, Berlin, 1977

[18] D.L. Parnas, *"A technique for the specification of software modules with examples"*, Comm. ACM, 15, pp. 330-336, 1972

[19] R. Reiter, On closed world data bases, in *"Logic in data bases"* (H. Gallaire and J. Minker ed's), pp. 55-76, Plenum Press, 1978

[20] J. Tardo, *"The design, specification and implementation of OBJ: A language for writing and testing abstract algebraic program specifications"*, PhD Thesis, UCLA, Comp. Sci. Dept., 1981

[21] J.W. Thatcher, E.G. Wagner, J. Wright, *Data type specification: parameterisation and the power of specification techniques,* Proc. ACM SIGACT 10th Symp. on Theory of Computing, pp. 119-132, 1978

[22] M. Wand, *Final algebra semantics and data type extensions,* Journal of Computer and System Science, 19, 1979, pp. 27-44

ESPERIENZE DI SVILUPPO DI UN SISTEMA INTERATTIVO PER IL PROGETTO CONCETTUALE DI ASPETTI DINAMICI DI BASI DI DATI

R. Bertocchi (*+), V. De Antonellis (*), R Giovacchini(*)
(*) Istituto di Cibernetica, Università degli Studi di Milano (+) Database Informatica S.p.A.

Summary

The aim of our research is to model database procedures in terms of events, and to execute the modeled procedures against a database. From the modeling point of view, we have developed a methodology and a computer aided system, INCOD-E, for the conceptual design of events.
In this paper the architecture of the system is described. INCOD-E supports the designer in the conceptualization of the procedures, automatically checking the consistency of the process and interacts with the designer in discovering conflicts, in prompting possible solutions and guiding him through the steps of the design activities. It is being implemented in a UNIX environment with INGRES database management system. Two UNIX tools, YACC and LEX, have been used for implementating the parser of the INCOD-E design language.
The research in progress is supported by the Italian National Council Of Research, Progetto Finalizzato Informatica - Obiettivo DATAID.

1. INTRODUZIONE

La nozione di schema concettuale, inteso come rappresentazione semantica di sistemi del mondo reale, sia per gli aspetti statici che per quelli dinamici, ha dato origine, nella recente letteratura sulle basi di dati, allo sviluppo di modelli che, oltre a descrivere la struttura degli oggetti, descrivano anche il loro comportamento mediante concetti quali procedure, operazioni, eventi, e flussi di controllo /2/,/3/,/5/,/7/, /8/,/13/,/18/,/22/,/24/.

In particolare la nostra ricerca concerne il modellamento e la gestione di procedure (attività del sistema oggetto) su una base di dati in termini di eventi e loro dipendenze. Con il termine "modellamento" intendiamo l'analisi e la descrizione di procedure. Con il termine "gestione" intendiamo l'esecuzione ed il controllo della esecuzione delle procedure. Con il termine "evento" intendiamo il cambiamento di condizioni che si verifica in un sistema in seguito all'esecuzione di una operazione; un evento è descritto dalle condizioni che devono valere prima dell'esecuzione di una operazione (pre-condizioni) e dall'operazione stessa. Una procedura è vista come una collezione di operazioni coordinate da condizioni, o, in altre parole, come un tessuto di eventi.

Per il modellamento di procedure abbiamo sviluppato un formalismo basato sulle reti di Petri /20/ e una metodologia che guida il progettista a partire dall'analisi delle procedure fino alla loro rappresentazione formale.

Il modellamento è effettuato nel seguente modo.

a) Descrizione delle procedure che devono essere modellate (Raccolta dei Requisiti).

b) Analisi delle procedure in termini di eventi (Analisi dei Requisiti). In particolare, dai requisiti utente, per ogni attività, vengono estratte quelle informazioni che fanno riferimento ad eventi per costruire frasi-eventi in linguaggio naturale ristretto. Una frase-eventi deve esprimere la conoscenza acquisita su un evento o su più eventi che condividono le pre-condizioni (blocco-eventi). Una frase-eventi, quindi, può essere composta da più condizioni che valgono concorrentemente o in alternativa, e da più operazioni da eseguire concorrentemente o in alternativa.

c) Associazione degli eventi a grafi elementari, del tipo reti di Petri, mediante la grammatica di associazione, e composizione di tali grafi elementari di tanti grafi (schemi-eventi) quante sono le procedure analizzate, mediante la grammatica di composizione (Modellamento di Viste).

d) Coordinamento degli schemi-eventi per ottenere uno schema-eventi globale che rappresenti la comunicazione fra le attività del sistema oggetto (Integrazione di Viste).

Per supportare la progettazione concettuale di procedure abbiamo sviluppato un sistema automatico, INCOD-E, che assiste il progettista di basi di dati nel disegno incrementale di schemi-eventi. INCOD-E fa parte del sistema INCOD-DTE (Interactive Conceptual Design of Data Transactions and Events) /2/,/3/ ed è stato sviluppato in stretta connessione con la metodologia DATAID-1 prodotta dall'obiettivo DATAID del Progetto Finalizzato Informatica del CNR.

2. IL MODELLO DI EVENTI

La conoscenza acquisita su eventi è specificata mediante quattro entità:

a) Il blocco-eventi con un nome che lo identifica ed è unico all'interno dello schema-eventi.

b) Le condizioni che devono valere perchè un evento possa verificarsi. Le condizioni hanno attributi che ne caratterizzano la struttura, la natura ed i rapporti causali.

L'attributo "tipo sintattico" specifica il tipo di composizione:

- AND/OR/OREX indicano la presenza di più condizioni che devono valere rispettivamente in congiunzione, alternativa non esclusiva, alternativa esclusiva.

Il "tipo pragmatico" classifica le cause del valore delle condizioni del blocco eventi:

- EXT indica condizioni che valgono come effetto di operazioni non comprese nell'attività cui fa riferimento la frase eventi;
- TRANS indica condizioni che valgono come effetto di operazioni proprie della attività cui fa riferimento la frase-eventi; tali condizioni fanno riferimento ai risultati di operazioni mediante variabili di stato;
- CAL indica condizioni che valgono in corrispondenza di situazioni di calendario.

Il "tipo semantico" distingue le condizioni in:

- condizioni temporali: WHEN
- condizioni causali: IF

ESEMPI DI CONDIZIONI

"Se nessuna delle richieste dell'utente è soddisfatta".
IF satisfy-travel-request. none-satisfied (TRANS)

"Quando arriva una richiesta o una modifica di viaggio".
WHEN MSG travel-request (EXT) OR WHEN MSG change-request (EXT)

"Quando è trascorso un mese dalla proposta alternativa di viaggio".
WHEN INTERVAL >1 month AFTER TRANS submit-proposal (CAL)

"Dopo il 31/12/83"
WHEN TIME > 31/12/83 (CAL)

c) Le operazioni, che debbono essere compiute perchè l'evento abbia luogo. Le operazioni sono caratterizzate da:

L'attributo "tipo sintattico" classifica la composizione:

- AND/OR/OREX indicano la presenza di più operazioni che possono essere eseguite in concorrenza, in alternativa non esclusiva, in alternativa esclusiva.

Il "tipo pragmatico" classifica la natura delle operazioni:

- operazioni di modifica della base di dati: MOD
- operazioni di osservazione della base di dati: OBS

ESEMPI DI OPERAZIONI:

"Completa la pratica"
define-dossier (MOD)

"Soddisfa la richiesta di viaggio"
satisfy-travel-request (OBS)

d) Il nome dell'attività cui fa riferimento il blocco eventi.

3. DESCRIZIONE FUNZIONALE DI INCOD-E

INCOD-E assiste il progettista di basi di dati nel disegno incrementale di schemi-eventi sia nella progettazione di viste locali che nella loro integrazione. INCOD-E consente verifiche di consistenza e di adeguatezza delle specifiche ad ogni passo della progettazione e, in caso di conflitti, interagisce con il progettista proponendo soluzioni; inoltre, facilita la gestione della documentazione e mantiene traccia della storia del progetto.

Le funzionalità di INCOD-E sono descritte schematicamente in fig. 1.

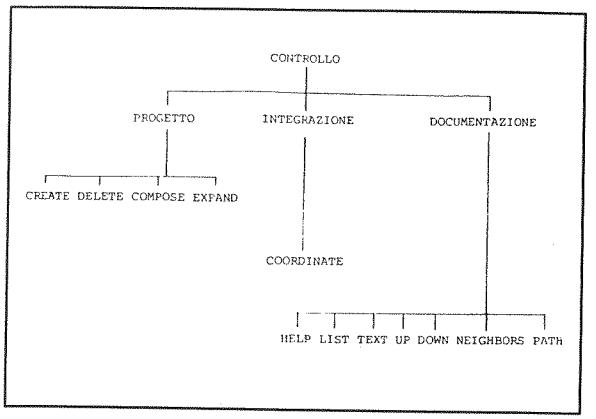


Fig. 1

La funzione di controllo assiste il progettista durante la fase di inizializzazione del progetto, di accesso e rilascio delle funzioni.

La funzione di progetto permette la costruzione incrementale dello schema eventi. I comandi a disposizione consentono di:

- creare un blocco eventi: E-CREATE;
- cancellare un blocco eventi: E-DELETE;
- comporre blocchi eventi in uno schema eventi. Gli eventi possono essere composti in sequenza, conflitto e concorrenza: E-COMPOSE;
- espandere un blocco eventi: E-EXPAND.

La funzione di integrazione permette di coordinare schemi eventi al fine di ottenere uno schema eventi globale.

La funzione di documentazione permette al progettista di ottenere informazioni di progetto riguardo a schemi già costruiti o anche a singoli blocchi eventi.

Nella fase di costruzione dello schema eventi possono prodursi conflitti nel modo di rappresentare l'informazione all'interno dello schema. Nell'ambito della nostra metodologia abbiamo classificato i conflitti in due classi principali: inconsistenze e inadeguatezze.

Una inconsistenza si verifica quando la rappresentazione contiene elementi in contrasto con le regole del modello adottato; ad esempio, nello schema esiste già un blocco eventi con lo stesso nome di quello che si vuole creare (inconsistenza di nome evento), oppure le operazioni del nuovo blocco compaiono in un altro blocco eventi della stessa attività (inconsistenza di nome operazione).

Una inadeguatezza si verifica quando la rappresentazione è consistente con il modello ma tuttavia non rispecchia ciò che si voleva rappresentare; ad esempio in un blocco eventi è definita una condizione che fa riferimento ad una operazione che non compare all'interno dello schema.

INCOD-E scopre i conflitti all'interno di uno schema e tra schemi diversi, ne dà segnalazione al progettista e presenta, ove possibile, scenari per la loro risoluzione.

Uno scenario propone interpretazioni del conflitto: se il progettista accetta una delle interpretazioni, viene attivata una sequenza di comandi che risolvono il conflitto. Per illustrare le modalità dell'interazione tra progettista e sistema è presentata in fig. 2 una sessione di lavoro in cui viene creato un blocco eventi che genera una inconsistenza di nome di operazione. Il progettista opera mediante maschere dinamiche, cioè maschere con le quali gli vengono richieste, passo per passo, le informazioni necessarie per soddisfare l'operazione richiesta.

(In figura, ciò che è digitato dall'utente è preceduto dal simbolo >).

```
› E-CREATE

EVENTS-BLOCK   : › satisfy;
ACTIVITY       : › booking;
CONDITIONS     : › WHEN MSG ISTANTIATE
                   booking (EXT)
OPERATIONS     : › satisfy-travel-request (OBS)

INCONSISTENCY  : satisfy-travel-request EXISTS
                 IN EVENTS-BLOCK accept
                 OF SCHEMA booking

› HELP

SCENARIOS:

1. DELETE THE PREEXISTENT EVENTS
   BLOCK AND DEFINE THE NEW ONE

2. MAINTAIN THE PREEXISTENT EVENTS
   BLOCK AND RENOUNCE TO THE NEW ONE

3. CHANGE THE NAME OF THE PREEXISTENT
   OPERATION

4. CHANGE THE NAME OF THE NEW OPERA-
   TION

5. CHANGE THE NAME OF BOTH OPERA-
   TIONS
```

Fig. 2

3. ARCHITETTURA DI INCOD-E

INCOD-E è composto di quattro componenti. In fig. 3 è mostrata la struttura del sistema: le linee continue rappresentano connessioni logiche tra i componenti; le linee tratteggiate rappresentano interazioni tra il progettista e il sistema, e le linee tratto e punto le connessioni con la base dati.

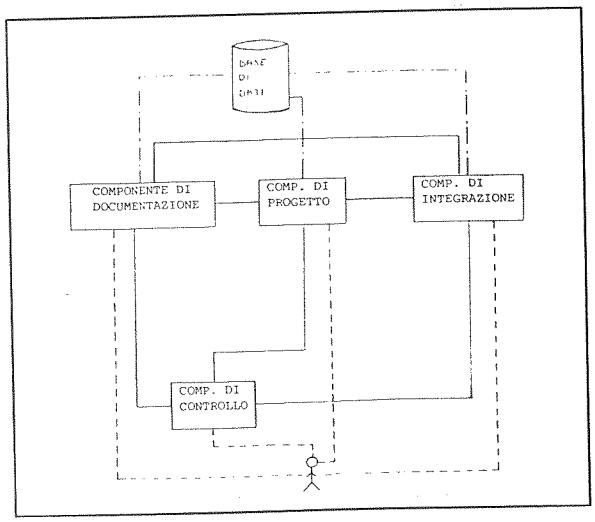


Fig. 3

I componenti di INCOD-E accedono ad una base di dati relazionale in cui vengono registrate le informazioni necessarie a classificare i blocchi eventi. In fig. 4 sono presentati esempi delle relazioni "condizione" e "operazione".

CONDIZIONE

codice	tipo sem.	espressione	tipo prag.
C1	IF	satisfy-travel-request. none satisfied	TRANS
C2	IF	satisfy-travel-request. all-satisfied	TRANS
C3	WHEN	MSG change-request	EXT
C4	WHEN	Time › 31/12/83	CAL

OPERAZIONE

codice	espressione	tipo prag.
T1	satisfy-travel-request	OBS
T2	define-dossier	MOD
	.....	.

Fig. 4

4. REALIZZAZIONE

INCOD-E si sta realizzando su VAX 11-780 con il sistema operativo UNIX; per la gestione della base di dati si è scelto il sistema INGRES di tipo relazionale.

Attualmente esiste una versione prototipale comprendente le componenti che realizzano le funzioni di controllo e di progetto. Il parser del linguaggio di progetto è stato sviluppato utilizzando due tools: LEX, un generatore automatico di analizzatori lessicali e YACC un generatore automatico di analizzatori sintattici.

Tali strumenti hanno favorito la facilità e rapidità di sviluppo dell'analizzatore sintattico del linguaggio di progetto, in quanto la conoscenza richiesta è relativa alle regole di costruzione dei comandi del linguaggio piuttosto che alle specifiche tecniche di parsing.
Inoltre è stato possibile seguire un approccio prototipale volto a realizzare, e allo stesso tempo verificare, parti del sistema in sviluppo indipendentemente dalla sua realizzazione completa.

Sulla base di queste considerazioni si è ritenuto opportuno descrivere nel seguito, brevemente, il funzionamento di questi due tools con riferimento al loro uso in INCOD-E.

4.1. LEX

Lex /17/ è uno strumento per la generazione automatica di analizzatori lessicali. Lex accetta in input un insieme di espressioni regolari (che costituiscono le regole di generazione delle stringhe di caratteri alfanumerici che devono essere riconosciute) con associate le azioni che l'analizzatore lessicale deve effettuare al riconoscimento di tali stringhe. Precisamente, Lex genera un programma C, chiamato convenzionalmente yylex, che, dato un testo di input, lo segmenta in stringhe che soddisfano le espressioni regolari definite, ed esegue le corrispondenti azioni. Quando yylex incontra caratteri che non soddisfano alcuna delle espressioni definite, yylex esegue un'azione di default che consiste nel copiare l'input nell'output (fig. 5).

Lex è stato usato insieme a Yacc per realizzare l'analizzatore lessicale (scanner) e l'analizzatore sintattico (parser) del linguaggio di progetto di INCOD-E.

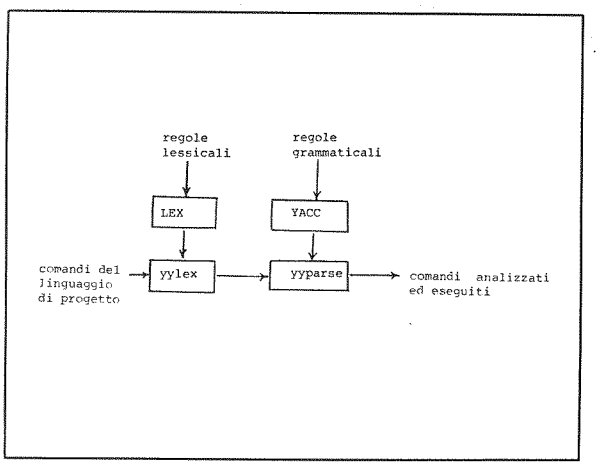


Fig. 5

L'input per Lex è specificato in un file comprendente tre sezioni:

dichiarazioni
%%
regole lessicali
%%
programmi

Tutte e tre le sezioni sono opzionali; se non viene specificata alcuna regola, yylex per qualsiasi carattere letto eseguirà solo l'azione di default di copiare l'input in output.

La sezione "dichiarazioni" è utilizzata per la dichiarazione di variabili che sono usate nelle azioni; le variabili dichiarate in questa sezione hanno scopo globale e quindi sono viste sia dalle azioni dell'analizzatore lessicale che dal parser che interfaccia Lex.

Nella sezione "regole lessicali" sono specificate, in forma tabellare, a sinistra le espressioni regolari che si vuole riconoscere e, per ognuna di esse, a destra le azioni che devono essere eseguite quando l'espressione viene riconosciuta (fig. 6).
Una espressione regolare è composta utilizzando caratteri di testo e operatori che specificano ripetizioni, alternative, raggruppamenti, etc.; più operatori possono essere utilizzati all'interno di una espressione. Ad esempio:

[A-Za-z] [A-Za-z0-9]*

è la tipica espressione usata per definire gli identificatori di un linguaggio dove [A-Za-z] indica un carattere alfabetico e [A-Za-z0-9]* una stringa di caratteri alfanumerici.

Le azioni sono costituite da una o più istruzioni in linguaggio C raccchiuse tra parentesi graffe.

Nella sezione "programmi" sono definite le subroutine utente (eventualmente ridefinizioni di routine standard di sistema) che possono essere richiamate all'interno delle azioni.

Nel nostro caso, utilizzando Lex assieme a Yacc occorre che, ad ogni regola di Lex che riconosce un token (simbolo terminale del linguaggio), sia associata l'azione: return (nome-del-token).

In questo modo quando una stringa soddisfa un'espressione del Lex l'analizzatore lessicale ritorna al parser il valore del token corrispondente mediante una costante simbolica (il nome del token) (fig. 6).

```
%%
.....
e-create                    return (CREATE);
e-delete                    return (DELETE);
"+"|"-"                     return (SEGNO);
[A-Za-z] [A-Za-z0-9]*       return (IDENTIFIER);
[0-9]+                      return (INT);
"("                         return (APERT);
.....
```

Fig. 6

Lex è in grado di gestire una specifica ambigua delle espressioni che devono essere riconosciute, cioè se in input viene riconosciuta una stringa che soddisfa due o più espressioni, la scelta viene effettuata seguendo, nell'ordine, due criteri:

1) è preferita l'espressione che incontra il più alto numero di caratteri;

2) a parità di caratteri, è preferita la regola che è stata definita per prima.

Esempio:

create	azione1	(1)
[a-z]*	azione2	(2)

se l'imput è "create" viene scelta l'espressione (1) in base al secondo criterio, se invece l'input è "create-event" viene scelta l'espressione (2) in base al primo criterio.

4.2. YACC

Yacc (Yet Another Compiler Compiler) /15/ è uno strumento che permette, data la grammatica di un linguaggio, di generare automaticamente l'analizzatore sintattico (parser).

L'utente che desidera utilizzare Yacc deve fornire in input:

a) le regole grammaticali del linguaggio;
b) le azioni che devono essere eseguite al riconoscimento di una regola del linguaggio (azioni);
c) la routine che fornisce all'analizzatore sintattico i simboli terminali, tokens, del linguaggio (analizzatore lessicale); questa routine può essere scritta

dall'utente o essere prodotta dal generatore automatico Lex.

In base alle regole grammaticali del linguaggio, Yacc genera un programma C che è chiamato convenzionalmente yyparse; yyparse chiamando ripetutamente la routine c) verificherà se il proprio input è conforme alle regole definite in a) e, per ogni regola soddisfatta, eseguirà le azioni corrispondenti b).

L'input per Yacc è specificato in un file comprendente tre sezioni:

dichiarazioni
%%
regole grammaticali
%%
programmi

Le sezioni "dichiarazioni" e "programmi" sono opzionali e se omesse, la specifica si riduce alle sole regole grammaticali.

La sezione "dichiarazioni" può essere utilizzata per la dichiarazione di variabili; le variabili definite in questa sezione hanno lo scopo globale e quindi sono viste sia dalle azioni del parser che dall'analizzatore lessicale.

La sezione "regole grammaticali" deve comprendere una o più regole con la seguente forma:

```
x  :a
   ;
```

dove: "x" è un simbolo non terminale;
"a" è una sequenza (anche vuota) di simboli terminali e non terminali. Il simbolo ":" indica la separazione tra la parte sinistra della regola grammaticale e la parte destra, il simbolo ";" indica la terminazione di una regola.

Più regole grammaticali con lo stesso simbolo non terminale a sinistra possono essere specificate utilizzando il carattere "!". Non esiste alcun vincolo sull'utilizzo delle lettere maiuscole o minuscole; la convenzione, che noi seguiremo nel testo, è che le prime indichino i tokens, le seconde i simboli non terminali (fig. 7).

I tokens devono essere obbligatoriamente dichiarati nella sezione "dichiarazioni", tutti i nomi non definiti in questa sezione sono interpretati da Yacc come simboli non terminali (eccetto quelli racchiusi tra apice) e, all'interno delle regole, dovranno essere successivamente ridotti a simboli terminali.

```
%%
token      FINE-SESSIONE CREATE DELETE
.....

%%
sessione          :lista-di-comandi FINE-SESSIONE
                  ;

lista-di-comandi  : /*vuota*/                      (1)
                  |comando lista-di-comandi
                  ;

comando           :creazione                       (2)
                  |cancellazione
                  |composizione
                  |espansione
                  ;

.....
```

Fig. 7

Nell'esempio di fig. 7, che fa riferimento al nostro linguaggio di progetto, una sessione è composta da uno o più comandi, ciascuno dei quali può essere uno di quelli definiti dalla regola (2).

Ad ogni regola grammaticale possono essere associate azioni specificate da istruzioni del linguaggio C; all'interno di un'azione possono essere richiamate subroutine C definite nella sezione "programmi" del file di input a Yacc o in un opportuno file utente.

Nel nostro caso, le azioni, oltre alle usuali istruzioni C, contengono anche istruzioni per accedere ad una base di dati. Queste istruzioni non facendo parte del C, ma del linguaggio di manipolazione della base di dati, devono essere prima pre-processate dall'interfaccia (EQUEL) esistente tra il DBMS ed il linguaggio C, e poi essere successivamente inserite nel file di specifiche.

Nella fig. 8 (alla pagina seguente) sono mostrate alcune regole grammaticali del linguaggio di progetto di INCOD-E e le azioni corrispondenti; per facilitare la lettura le istruzioni per accedere alla base di dati sono presentate in forma non processata.

```
%%
.....
char ename [20];
struct cond {
        char e-cond [20]
        char e-name [20]
        char t-sem [20]
        char t-prag [20]
        } a [10];

.....
%%
creazione     : CREATE                          (1)
                        mask-create ( );
                nome-evento attività condizioni ope-
                razioni
                {
                  char *ind [5]
                .....
                ind [0] = e-cond;
                ind [1] = e-name;
                .....
                ingres incod:          /* ACCESSO
                param append
                to rel-condizioni      ALLA BASE
                ("e-cond, e-name,
                t-sem, t-prag", ind)       DATI  */
                exit
                  .....
                }
cancellazione : DELETE                          (2)
                        mask-delete ( );
                nome-evento nome-attività
                ;
nome-evento   : IDENTIFIER ';'                  (3)
                {
                verifica-incons ( );
                verifica-inadeg ( );
                .....
                .....
                strcpy (ename, yytext);
                .....
                }
                ;
attività      : IDENTIFIER ';'                  (4)
                {
                verifica-incons ( );
                verifica-inadeg ( );
                .....
                .....
                strcpy (eattiv, yytext);
                .....
                }
                ;
condizione    : body ';'                        (5)
                ;
body          : elementare                      (6)
                | APERT body operats
                  body CHIUSA
                ;
elementare    : SE stringa PUN EX EQ bool
                QUADA TR QUADC                  (7)
                | QUANDO MES stringa
                  QUADA EXT QUADC
                  .....
                  .....
                ;

.....
.....
%%
mask-create ( )
.....
verifica-incons ( )
.....
verifica-inadeg ( )
.....
.....
```

Fig. 8

La regola (1) definisce la sintassi del comando per la creazione di eventi, il token CREATE viene ritornato quando l'utente digita il comando e-create; quando il comando è riconosciuto viene presentata una maschera per l'inserimento delle entità che definiscono l'evento da creare; le entità sono definite in (1) come i simboli non terminali: nome-evento, attività, condizioni, operazioni.

Supponiamo che il progettista abbia scritto:

EVENTS-BLOCK : › satisfy-request;

ACTIVITY : › agency-booking;

CONDITIONS : › (WHEN MSG travel-request (EXT) OR WHEN MSG change-request (EXT));

OPERATIONS : › satisfy-travel-request;

Nome evento ed attività sono definiti in (3) e (4) come il token IDENTIFIER seguito dal carattere ';', le azioni che vengono invocate in queste regole sono due routine che verificano l'esistenza di inconsistenze e inadeguatezze.

Nell'esempio la specifica dell'utente è corretta rispetto alle regole (3) e (4) e, supponendo che non siano verificate inconsistenze o inadeguatezze, la stringa contenente il nome dell'evento (yytext) viene copiata nella variabile ausiliaria ename; analogamente, il nome dell'attività viene copiato nella variabile e attiv. Mediante le regole (5) (6) (7) viene verificato se la condizione (WHEN MSG travel-request (EXT) OR WHEN MSG change-request (EXT)) è corretta, e così via fino ad esaurimento della regola (1).
Infine le informazioni relative al blocco eventi vengono copiate nell'array ind e successivamente registrate nelle relazioni della base di dati.

Nell'ambito della costruzione di un parser, la gestione degli errori è un'attività che presenta notevoli difficoltà e comporta particolare attenzione nel decidere il comportamento ottimale che il parser dovrà seguire quando viene rilevato un errore. Per permettere all'utente di controllare il processo di gestione degli errori, Yacc fornisce un token di nome "error" che può essere utilizzato, nel contesto della grammatica, per specificare dove possono prodursi degli errori e quali operazioni di recovery devono essere effettuate.

L'utente, in caso di errore, ha due possibilità per ripristinare lo stato del parser e continuare l'analisi:

- affidarsi a procedure standard di error-recovery;
- specificare un simbolo di reinizializzazione che ha il significato di comunicare al parser di ignorare l'input compreso tra il token che ha provocato l'errore e il simbolo di reinizializzazione riprendendo l'analisi dal token successivo a tale simbolo.

Nel nostro caso, trattandosi di una applicazione interattiva, oltre a definire un simbolo di reinizializzazione, il parser è stato predisposto ad accettare una correzione immediata dell'errore.

```
nome-evento   :error ';'
              | yyerrok;

              clear mask ( );
              .....
              for  (i=0; i<10; i++)
                   {strcpy (a [i].e-cond, "\0");
                    strcpy (a [i].e-name, "\0");
                    strcpy (a [i].t-sem, "\0");
                    strcpy (a [i].t-prag, "\0");
                   }
              .....

              nome-evento
              |IDENTIFIER ';'
              ;
              .....
              .....
```

Fig. 9

Nell'esempio di fig. 9 si richiede che il parser, in presenza di un nome-evento scorretto, ignori l'input successivo fino al carattere ';', esegua le azioni specificate (reinizializzazione di variabili, posizionamento del cursore etc.) e, al termine, resti in attesa di un nuovo nome-evento.

Per soddisfare esigenze più sofisticate l'utente può definire delle proprie routine di risincronizzazione e, se desidera una diagnostica più ricca di quella standard, può introdurre, a fianco della segnalazione d'errore, altre informazioni quali il token che lo ha provocato, il suo numero di linea nel file di input, etc.

Infine va segnalato che, quando una regola grammaticale è ambigua, cioè una stringa di input può essere generata da due o più regole grammaticali, Yacc produce egualmente il parser utilizzando delle regole di default che eliminano le ambiguità. Situazioni di ambiguità possono prodursi sia a causa di effettivi errori compiuti nella specifica della grammatica, sia perchè le regole grammaticali, seppur corrette, richiedono un parser più complesso di quello che Yacc può generare. In questi casi Yacc genera il parser riportando il numero di conflitti risolti e le regole di default che ha adottato; l'utente deve verificare che la risoluzione dei conflitti sia conforme al comportamento del parser che egli desiderava.

Per una descrizione dettagliata del funzionamento del parser e degli algoritmi usati per la sua generazione rimandiamo agli articoli /1/ /15/.

5. CONCLUSIONI

In questo lavoro è stata presentata l'architettura di un sistema interattivo, INCOD-E, di ausilio alla progettazione concettuale di procedure su basi di dati.

In particolare sono stati illustrati aspetti di realizzazione in ambiente Unix relativi all'analisi lessicale e sintattica del linguaggio di progetto. In questo ambito i principali problemi che si stanno indagando sono relativi alla gestione degli errori, sia di tipo sintattico, allo scopo di superare alcune delle limitazioni proprie dello Yacc, sia di tipo semantico allo scopo di definire meccanismi di controllo delle inconsistenze che risultino efficienti ed adeguati in un contesto interattivo.

Estensioni dell'attuale versione prototipale del sistema prevedono la realizzazione della funzionalità di integrazione e documentazione, e lo studio di una interfaccia grafica che accompagni interattivamente l'attività del progettista.

BIBLIOGRAFIA

(1) Aho A.V., Ullmann J.D. *Principles of Compiler Design,* Addison-Wesley, Reading, Mass, 1977

(2) Atzeni P., Batini C., Lenzerini M., Villanelli F. *INCOD- DTE: a system for conceptual design of data and transactions in the entity-relationship model,* Methodology and Tools for Database Design, S. Ceri ed. North Holland, 1983

(3) Atzeni P., Batini C., De Antonellis V., Lenzerini M., Villanelli F., Zonta B., *A Computer aided tool for conceptual data base design,* Proc. of the IFIP WG 8.1 Working Conf. on Automated Tools for Information Systems Design and Development, New Orleans, 1982

(4) Barrow J.L., *Dialogue organization and structure for interactive information systems,* TR-GSRG-108, Computer Systems Research Group, University of Toronto, 1080

(5) Bertocchi R., De Antonellis V., Zonta B., *Concepts and mechanisms for handling dynamics in database applications,* 7th International Computing Symposium, ACM European Regional Conference, March 1983

(6) Breutman B., Falkenberg E., Mauer R., *CSL: a language for defining conceptual schemas,* in IFIP Working Conference on DB Architecture, 1979

(7) Brodie M., Silva E., *Active and passive component modelling: ACM/PCM,* Information System

Design Methodologies: a comparative review, North Holland, 1983

(8) Bubenko J.A. *The temporal dimension in information modelling,* in Architecture and Models on DBMS, G.M. Nijssen ed., North-Holland Pub. Co., 1977

(9) Bussolati U., Ceri S., De Antonellis V., Zonta B., *Views Conceptual Design*, Methodology and Tools for Database Design, S. Ceri ed., North Holland, 1983

(10) Chen P.P., *The entity relationship model: toward a unified view of data,* ACM Transaction on Data Base Systems, Vol. 1.1, 1976

(11) De Antonellis V., Zonta B., *Modelling Events in data base application design,* Proc. Int. Conf. on Very Large Data Bases, Cannes, 1981

(12) De Antonellis V., Demo B., *Requirements collection and analysis,* Methodology and Tools in Database Design, S. Ceri ed., North Holland, 1983

(13) Hammer M., Mc Leod D., *Database description with SDM: a semantic database model,* ACM TODS 1981, Vol. 6

(14) Infotech Ltd, *An introduction to data structure system design,* 1980

(15) Johnson S.C., *Yacc - Yet Another Compiler,* manuale Unix

(16) Kernighan B.W., Ritchie D.M., *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978

(17) Lesk M.E., *Lex - A Lexical Analyzer Generator,* manuale Unix

(18) Mylopoulos J., Berstein P., Wong H., *A language facility for designing database intensive application,* ACM TODS 1980, Vol. 5

(19) Navathe S.B., ed altri, *Information modelling tools for data base design,* data base directions, Fort Lauderalde, Florida, 1980

(20) Petri C.A., *Introduction to general net theory,* in Lecture Notes in Computer Sciences, n. 84, Springer-Verlag, 1980

(21) Ritchie D., Thompson K., *The Unix Time-Sharing System,* Comm. Assoc. Comp. Mach. 17 (7), 1974

(22) Rolland C., Richard C., *The REMORA methodology for information systems design and management,* Information System Design Methodologies: a comparative review, North Holland, 1983

(23) Stonebraker M., Wong E., Kreps P., *The design and implementation of INGRES,* ACM Trans. Database Systems 1 (3), 1976

(24) Tardieu H., Nanci D., Pascot D., *Conception d'un sisteme d'information,* Les Editions de l'Organisation, Paris 1980

(25) Zisman M.D., *Use of production systems for modelling asynchronous concurrent processes,* in Pattern Directed Inference Systems, Academic Press, 1978.

BIBLIOGRAFIE

GLI EXPERT SYSTEMS

Stefania Bandini – Istituto di Cibernetica – Università di Milano
Massimo Gallanti – C.I.S.E. – Segrate

Quello sugli EXPERT SYSTEMS è senz'altro uno degli argomenti di punta di questi ultimi mesi. Tale interesse si è manifestato anche in Italia attraverso una proliferazione di dibattiti, corsi, seminari, convegni e tavole rotonde dove la gamma dei partecipanti è delle più eterogenee: l'industria ha marcato la sua presenza con la stessa curiosità della ricerca universitaria, a fianco delle grandi organizzazioni a partecipazione statale e le giovani, ma non meno aggressive, software houses.
Questa ampia convergenza di interessi non risponde ad un tacito richiamo dettato da sete teorica, ma trova la sua sorgente nel successo applicativo (e dunque di mercato) di sistemi costruiti con una metodologia particolare, quella che risale agli strumenti teorici delle Intelligenze Artificiali.

Gli Expert Systems sono sistemi basati sulla conoscenza: la rappresentazione della conoscenza, infatti, è un tema che svolge le sue note sullo studio delle tecniche per rappresentare in forma simbolica conoscenze di un settore e per sviluppare su di esse attività inferenziali e di ragionamento complesse, addirittura come prerogative essenzialmente umane.

Più esplicitamente, un Expert System cattura le conoscenze specifiche che un esperto umano accumula nella sua attività in un settore ben definito e fornisce consulenze specialistiche.

Infatti, fallito in parte l'antico sogno delle Intelligenze Artificiali di poter costruire un General Purpose System o un risolutore generale di problemi, le ricerche si sono indirizzate verso settori più delimitati e più controllabili del sapere umano.
Dal punto di vista applicativo, sono stati costruiti degli Expert Systems che lavorano su campi ristretti (specializzazione di applicazioni mediche, settori della ricerca geologica, etc.) e i risultati positivi hanno dato una motivazione alla spinta di interessi a questo modo di usare sistemi informativi.

Ciò che in questa sede si vuole fornire è una nota bibliografica generale sugli Expert Systems; tali suggerimenti sulla documentazione della materia non hanno pretese di esaustività, ma hanno lo scopo di tracciare alcuni punti di riferimento all'interno di una letteratura non vastissima (data la giovane età della disciplina), ma ancora frammentaria.

La linea di approccio bibliografico che qui si vuole suggerire contempla una prima parte di carattere generale sulle ricerche teoriche delle Intelligenze Artificiali che selezionano indirizzi verso la realizzazione di Expert Systems, mentre ad un secondo gruppo di documenti è affidato il compito di entrare nello specifico di una letteratura sulle realizzazioni compiute. Una terza parte, infine, sarà dedicata all'illustrazione tassonomica per argomenti della letteratura riguardante le applicazioni specifiche degli Expert Systems.

Letteratura generale

Le indicazioni bibliografiche di carattere generale che qui si forniscono non sono orientate particolarmente agli Expert Systems, ma fanno parte del patrimonio generale della letteratura delle Intelligenze Artificiali. In queste opere si possono trovare gli strumenti teorici elaborati durante l'arco degli ultimi decenni che hanno dato corposità alla disciplina.
Per arrivare agli Expert Systems è indispensabile un approccio a tali letture e la prova di questo sta nel fatto che, anche nella letteratura più specifica, non viene mai a mancare una nota introduttiva a queste metodologie che fanno delle Intelligenze Artificiali un settore ben definito tra le discipline informatiche.
L'elenco, ovviamente, non è esaustivo: si è pensato di suggerire sono alcune indicazioni che possono soddisfare le esigenze di un approccio introduttivo. Ognuna delle opere elencate presenta un carattere prevalentemente manualistico ed in nessuna di esse mancano ulteriori approfondite proposte bibliografiche per arricchire interessi specifici.

A. Barr, E.A. Feigenbaum (Eds), THE HANDBOOK OF ARTIFICIAL INTELLIGENCE, 3 voll., William Kaufmann Inc., Los Altos, CA, 1981.
È un opera suddivisa in tre volumi: il primo comprende una vasta panoramica sulle Intelligenze Artificiali e tratta in modo diffuso e particolareggiato delle metodologie per la rappresentazione della conoscenza. Il secondo volume vede brevemente illustrate le realizzazioni più note e storicamente più rilevanti dei Sistemi Esperti, classificandoli per campi d'appli-

cazione: in questa prassi descrittiva vengono messe in risalto le strategie di risoluzione dei problemi incontrati nei settori applicativi, più che gli aspetti ingegneristici e tecnologici; un intero capitolo è dedicato ai linguaggi di programmazione più adatti in questo settore.
Il terzo volume completa l'esposizione del volume precedente e passa ad una rassegna dei problemi relativi all'apprendimento e alla visione.

N.J. Nillson, PRINCIPLES OF ARTIFICIAL INTELLIGENCE, Tioga, Palo Alto, CA, 1980.
È uno dei "classici" della letteratura delle Intelligenze Artificiali; non esamina specificatamente i Sistemi Esperti, ma tratta in modo dettagliato della rappresentazione della conoscenza. Di particolare interesse è il capitolo dedicato alle metodologie basate su regole di produzione, che sono quelle maggiormente utilizzate nei Sistemi Esperti finora realizzati.

P.H. Winston, ARTIFICIAL INTELLIGENCE, Addison-Wesley, Reading, MA, 1977.
Un altro "classico" che mette in particolare risalto i problemi di identificazione di forme geometriche e in generale quelli della visione. I capitoli V e IX sono i più rilevanti rispetto alle tematica dei Sistemi Esperti.

P.H. Winston, R.H. Brown (Eds), ARTIFICIAL INTELLIGENCE. AN MIT PERSPECTIVE, MIT Press, Cambridge, MA, 1979.
Diviso in due volumi, è una rassegna di capitoli che toccano tutti i problemi delle Intelligenze Artificiali e in particolar modo della visione. Agli Expert Systems è dedicato il primo capitolo del primo volume.

Letteratura specifica.

Dell'interesse suscitato dai sistemi esperti si trova riscontro in letteratura, dove, a fianco di articoli specializzati che illustrano nel dettaglio la realizzazione di particolari sistemi, è abbastanza comune ormai imbattersi in scritti di rassegna generale, dai quali è possibile avere una panoramica di quanto è stato sviluppato nel settore dei Sistemi Esperti.

D. Mitchie (Ed), EXPERT SYSTEMS IN THE MICRO ELECTRONIC AGE, Edinburgh University Press, 1979.
È una collezione di articoli tratti da una scuola estiva tenutasi ad Edinburgo. Lo spettro dei problemi trattati si estende da articoli che illustrano le tematiche concernenti la realizzazione di Sistemi Esperti fino ad una riflessione epistemologica della disciplina.

M. Stefik et al., THE ORGANIZATION OF EXPERT SYSTEMS: A PRESCRIPTIVE TUTORIAL, Xerox, Palo Alto, CA, 1982.
È un articolo/guida alla realizzazione di Sistemi Esperti. In una panoramica teorica delle Intelligenze Artificiali, si limita solo alla descrizione di quegli strumenti concettuali utili alla costruzione di Expert Systems. La descrizione dei Sistemi Esperti realizzati, a cui è dedicata la seconda parte del lavoro, viene fornita mediante una classificazione basata sui vincoli a cui è soggetta la rappresentazione della conoscenza in ogni particolare realizzazione.

N. Butchnan, NEW RESEARCH ON EXPERT SYSTEMS, "Artificial Intelligence Magazine", n. 10, 1982.
È un lavoro che affronta la tematica dei sistemi esperti illustrando i punti meno assestati della disciplina e proponendo obiettivi da perseguire negli anni a venire, senza entrare in merito a problemi realizzativi.

AA.VV., A NEW TECHNICAL STUDY: ARTIFICIAL INTELLIGENCE – A NEW TOOL FOR INDUSTRY AND BUSINESS, Technical Insight Inc., Fort Lee, NY, (pagg. 159, $520).
Dei cinque capitoli più appendice che lo compongono, il primo fa un introduzione storica e teorica alle Intelligenze Artificiali, il secondo tratta dei Sistemi Esperti costruiti finora, raggruppandoli per settori d'applicazione, e gli altri tre capitoli illustrano problemi strettamente realizzativi: di costo, di mercato e di centri di produzione ("Industrial AI"). L'appendice accenna ai progetti in corso nei maggiori centri di ricerca americani.

D.S. Nau, EXPERT COMPUTER SYSTEMS, "Computer", Feb. 1983.
La prima parte dell'articolo illustra alcuni concetti generali riguardanti il "problem solving". L'autore poi esamina le tematiche dei Sistemi Esperti con l'aiuto di alcuni esempi. Sono riportate infine alcune considerazioni sugli investimenti necessari per la realizzazione di un Sistema Esperto.

G. Guida, C. Tasso, IL GRANDE FRATELLO?, "Sistemi e Automazione", n. 230, Ott. 1982.
Articolo che fa un panoramica sulle problematiche delle Intelligenze Artificiali che hanno trovato risonanza nel mondo industriale, in particolare per quel che riguarda l'ingegneria della conoscenza, della quale l'autore traccia le direttive principali, con l'ausilio di alcuni esempi. Viene inoltre fornita l'indicazione di enti industriali che sono impegnati in questo campo. L'articolo si chiude con una sintetica ma esaustiva bibliografica sulle Intelligenze Artificiali.

F. Gardin, INTELLIGENZA ARTIFICIALE E SISTEMI ESPERTI, "BIT", n. 32, n. 34, 1982, n. 36, 1983.
Tre articoli comparsi recentemente su un periodico di carattere divulgativo. Questi lavori, rivolti ad un pubblico di non addetti ai lavori, illustrano la struttura e le caratteristiche dei Sistemi Esperti. Vengono descritte in breve le realizzazioni più famose e, per chi vuole occuparsi dell'argomento in modo più diretto, sono riportate indicazioni su quali prodotti di svi-

luppo per Sistemi Esperti sono disponibili attualmente sul mercato.

Applicazioni

Nei lavori che qui si propongono l'accento viene posta sulla descrizione delle realizzazioni di Expert Systems nei loro settori applicativi. La struttura espositiva è basata su una classificazione per argomenti: di ogni sistema viene riportata la pubblicazione più significativa elaborata dagli autori del sistema stesso. Il lettore potrà incorrere in difficoltà nel reperimento di alcuni di questi documenti, trattandosi di rapporti interni di università americane o di Congressi: le fonti vengono qui ugualmente citate; va per altro detto che informazioni di massima su gran parte di questi sistemi sono riportate anche nella letteratura più generale sopra descritta (ad es. THE HANDBOOK OF ARTIFICIAL INTELLIGENCE).

Medicina:

MYCIN Realizzato per l'identificazione delle infezioni batteriche e delle terapie da adottare per combatterle.
E.H. Shortliffe, COMPUTER BASED MEDICAL CONSULTATION: MYCIN, American Elsevier, New York, 1976.

TERESIAS È un supporto che permette l'acquisizione automatica della conoscenza, mediante dialogo, di nuove regole per MYCIN.
R. Davis, D. Lenat, KNOWLEDGE BASED SYSTEMS IN ARTIFICIAL INTELLIGENCE, Mc Graw Hill, New York, 1982.

INTERNIST Sistema di consultazione nel campo della medicina interna.
H. Pople, J. Myers, THE FORMATION OF COMPOSIT; A HYPOTESES IN DIAGNOSTIC PROBLEM SOLVING: AN EXERCISE IN SYNTHETIC REASONING, IJCAI 5.

PUFF Si tratta di un sistema che fornisce l'interpretazione della funzionalità polmonare.
J. Kunt et al., A PHISIOLOGICAL RULE-BASED SYSTEM FOR INTERPRETING PULMONARY FUNCTION TEST RESULTS, Heuristic Programming Project, Rep. n. HPP/78/19, Computer Science Department, Stanford University, 1978.

CASNET Sistema di diagnosi rivolto al campo dell'oculistica: permette l'identificazione di glaucomi.
S. Weiss et al., A MODEL BASED METHOD FOR COMPUTER AID MEDICAL DECISION MAKING, "Artificial Intelligence", 11, 1978.

VM Il sistema permette il monitoraggio in tempo reale di un paziente sottoposto a terapia intensiva (polmone d'acciao).
L. Fagan et al., KNOWLEDGE ENGINEERING FOR DYNAMIC CLINICAL SETTINGS: GIVING ADVICE IN THE INTENSIVE CARE UNIT, Doctoral Dissertation, Computer Science Dept., Stanford University, 1979.

LITO 1 Un sistema esperto per la valutazione delle funzionalità epatiche sviluppato presso l'istituto di Scienze dell'Informazione dell'Università di Torino.
P. Torasso et al., AN EXPERT SYSTEM FOR THE EVALUTATION OF LIVER FUNCTIONAL ASSESMENT, Proc. of the First Annual Conference of AAMS, Washington, 1982.

Chimica:

DENDRAL È usato per l'identificazione di composti organici.
G. Buchnan, E. Feigenbaum, DENDRAL AND META-DENDRAL: THE APPLICATION DIMENSION, "Journal, of Artificial Intelligence", 11, 1978.

CRYSALIS Permette l'identificazione della struttura delle proteine utilizzando i dati provenienti da cristallografie.
R. Engelmore, H. Nil, STRUCTURE AND FUNCTION OF CRYSALIS SYSTEM, IJCAI 6.

SECS Viene utilizzato per la progettazione di sintesi organiche.
W. Wipke et al., SECS: SIMULATION AND EVALUATION OF CHEMICAL SYNTHESIS: STRATEGY AND PLANNING, in W. Wipke and W. Hause Eds., "Computer assisted organic synthesis", D.C. American Society, Washington, 1977.

Geologia:

PROSPECTOR Fornisce analisi su dati ricavati da prospezioni minerarie.
R. Duda et al., DEVELOPMENT OF THE PROSPECTOR CONSULTATION SYSTEM FOR MINERAL EXPLORATION, Final Report, SRI Projects, Menlo Park, CA, 1978.

DIPMETER ADVISOR È utilizzato nella ricerca di giacimenti petroliferi.
R. Davis et al., THE DIPMETER ADVISOR: INTERPRETATION OF GEOLOGICAL SIGNALS, IJCAI 7.

Didattica:

SOPHIE È un sistema che guida lo studente nell'apprendimento di metodologie per la progettazione di circuiti elettronici.
D. Sleeman, J. Brown (Eds), INTELLIGENT TUTORING SYSTEMS, Academic Press, London, 1983.

GUIDON Utilizza la base di conoscenza di MYCIN (malattie infettive del sangue), orientato, però, all'insegnamento delle metodologie di diagnosi piuttosto che alla diagnosi in se stessa.
W. Clancey, TUTORING RULES FOR GUIDING A CASE METHOD DIALOGUE, "International Journal of Man-Machine Studies", n. 11, 1979.

Ingegneria:

SACON Fornisce consulenze agli ingegneri sull'uso di procedure per l'analisi di strutture.
J. Bennet, R. Engelmore, SACON: A KNOWLEDGE-BASED CONSULTANT FOR STRUCTURAL ANALYSIS, Proceedings IJCAI, 1979.

Meccanica razionale:

MECHO È un sistema che risolve problemi di meccanica razionale.
A. Bundy, L. Byrd, G. Luger, C. Mellish, M. Palmer, SOLVING MECHANICS PROBLEMS USING META-LEVEL INFERENCE, in "Expert Systems in the Micro Electronic Age". D. Michie (Ed), Edinburgh University Press, Edinburgh, 1979.

Diagnostica industriale:

DART Nato da una collaborazione tra IBM e l'Università di Stanford, fornisce tecniche avanzate per la diagnosi di guasti in sistemi di calcolo: è applicato al sistema IBM 43/41.
J. Bennet, C. Hollander, DART: AN EXPERT SYSTEM FOR COMPUTER FAULT DIAGNOSIS, Proceedings IJCAI, 1981.

IDT Si tratta di uno studio sulle metodologie da applicarsi nella diagnosi dei sistemi elettronici. È stato sviluppato in ambito Digital.
H. Shubin, R. Hal, J. Ulrich, IDT: AN INTELLIGENT DIAGNOSTIC TOOL, Proceedings of the National Conference on Artificial Intelligence, 1982.

DELTA-CATS Sistema sviluppato presso la General Electric che si occupa della ricerca di guasti e la riparazione di grandi motori diesel, ad es. locomotive.
P. Bonissone, OUTLINE OF THE DESIGN AND IMPLEMENTATION OF A DIESEL ELECTRIC ENGINE TROUBLESHOOTING AID, Proceedings Expert Systems Conference, London, 1982.

SPERIL Sistema per la valutazione di danni a strutture statiche che sono state soggette a sollecitazioni (es. terremoti); utilizza informazioni rilevate da ispezioni visive effettuate in vari punti della struttura.
M. Ishizuka, K. Fu, J. Yao, SPERIL: AN EXPERT SYSTEM FOR DAMAGE ASSESSMENT OF EXISTING STRUCTURES, Proceedings of the Sixth International Conference on Pattern Recognition, Munchen, Ott. 1982.

Configurazione di calcolatori:

R1 È utilizzato per configurare il calcolatore VAX 11/780 partendo dai moduli di base che lo compongono. Questo sistema è stato svilupato presso la Carnegie-Mellon University per conto della Digital Equipment Corporation.
J. McDermott, R1: A RULE-BASED CONFIGURER OF COMPUTER SYSTEMS, Carnegie-Mellon University Press, 1980.
J. McDermott, R1: THE FORMATIVE YEARS, Artificial Intelligence Magazine, 2, 1980.

XSEL Realizza un front-end di R1; coadiuva il venditore per fornire una configurazione hardware/software per il VAX 11/780 sulla base delle esigenze del cliente.
J. McDermott, XSEL: A COMPUTER SALES PERSON'S ASSISTANT, Artificial Intelligence Magazine.

Matematica:

AM È un sistema per la scoperta automatica di teoremi.
R. Davis, D. Lenat, KNOWLEDGE BASED SYSTEMS IN ARTIFICIAL INTELLIGENCE, McGraw Hill, New York, 1982.

RECENSIONI

"Software Engineering - Analysis and Verification", T.G. Lewis, Reston Pub. Co., Reston, Virginia, 1982, pp. x + 470.

"L'idea che un essere umano possa formalmente dimostrare che un programma è corretto riscrivendolo come un sistema di asserzioni mi sembra sbagliata: perchè la logica dei predicati dovrebbe essere più efficace del linguaggio di programmazione stesso?" Supponiamo che abbiate in mano una copia del libro di Lewis e abbiate appena letto questa frase della prefazione: avete qualche esperienza di dimostrazioni formali di correttezza, e vi affrettate quindi a esaminare il capitolo 3 "A Transform Theory of Software Verification". Scorrete con impazienza le prime quattro pagine sul testing e arrivate finalmente al paragrafo sulle "trasformazioni funzionali delle componenti del software". Si comincia dagli assegnamenti (pag. 84): superate una piccola difficoltà (s_i è un assegnamento o una sequenza?) e vi scontrate con una difficoltà più grossa: l'esecuzione di una specifica sequenza di assegnamenti è equivalente all'esecuzione in parallelo di quegli stessi assegnamenti, ossia indipendente dall'ordine?! Poi un esempio vi chiarisce che le istruzioni di assegnamento da eseguire in parallelo non sono le stesse di quelle da eseguire in sequenza, e capite che l'equivoco è nato da un errore di notazione. Naturalmente le cose non funzionano se la stessa variabile compare a destra e a sinistra dell'assegnamento: occorre introdurre degli indici e tutto si complica. Quando arrivate in fondo e scoprite che per le strutture condizionali l'autore propone sostanzialmente di provare tutti i cammini possibili, siete alquanto delusi.

Purtroppo quanto abbiamo delineato per il capitolo 3 è il paradigma comune almeno ai primi capitoli, che ho esaminato più in dettaglio: alcuni approcci appaiono interessanti e nuovi, ma si rivelano poi piuttosto deludenti. Prendiamo il capitolo in cui si propone l'uso delle reti di Petri come modello dell'esecuzione di un programma, allo scopo di valutarne le prestazioni. L'idea sembra promettente, anche se le difficoltà intriseche all'analisi degli algoritmi si presentano ben presto, ma perchè non portare avanti coerentemente l'idea introducendo le marcature e rappresentando le diramazioni possibili come archi uscenti da uno stesso posto? L'autore invece non introduce le marche e usa quindi le reti come una rappresentazione puramente statica, arbitrariamente associando le suddivisioni di flusso alle transizioni. Introduce poi la trasformata geometrica in maniera alquanto oscura, complicata anche dai frequenti errori di stampa (specialmente omissioni o intrusioni di simboli nei diagrammi).

Per motivi che mi sfuggono i riferimenti bibliografici sono assai aleatori: quattro capitoli ne sono privi, e per esempio non c'è un singolo riferimento sulla trasformata geometrica o i B-alberi, e neppure sulle reti di Petri. Gli unici lavori citati a proposito di metodi di ricerca, ordinamento e hashing sono un libro di Wirth e un articolo e un libro dell'autore. Ma il capitolo sull'hashing merita un breve discorso a parte.

Anche qui il contenuto appare promettente: un metodo originale di hash perfetto, ossia senza collisioni. Il metodo è in effetti molto semplice e si presta bene a un'introduzione all'hashing a scopo didattico, anche perchè l'analisi dell'algoritmo è essa stessa non difficile; poichè però l'autore fa delle affermazioni non ben giustificate sui limiti di variazione dei parametri, il lettore è invogliato a rifare i conti e scoprire che i limiti forniti sono effettivamente errati. Ma, a parte questo, un diagramma riportato nel testo mostra come il fattore di carico (rapporto tra posizioni usate e posizioni richieste) di quel metodo di hashing sia intorno al 50% per 10 chiavi e sotto il 10% per 50 chiavi: in effetti qualche sperimentazione fatta mi induce a ritenere che quel diagramma sia addirittura largamente ottimistico. Il guaio è che quello stesso metodo di hashing è stato pubblicato, insieme con un altro più complesso, in un articolo del Prof. Sprugnoli sulle Communications ACM del novembre 1977: mi sembra scorretto non accennarvi neppure (tanto più che l'articolo di Sprugnoli è l'unico che ho trovato che citi a sua volta un lavoro di Lewis!) (*). Inutile negare quindi che la mia impressione sul libro non è del tutto positiva, tuttavia ribadisco che vi sono alcune idee interessanti, e che anzi proprio le manchevolezze possono stimolare il lettore attento.

Un punto a favore del libro è l'ampio spettro degli argomenti trattati. C'e un capitolo 0 che presenta una rassegna di modelli sul ciclo di vita del software, ricco di formule e tabelle tratte dalla letteratura (qui i riferimenti sono abbondanti). Questo capitolo non viene ripreso nel seguito del volume, ma ha lo scopo di rendere consapevole il lettore delle principali problematiche connesse al ciclo di vita e di evidenziare di conseguenza la necessità di tecniche di ingegneria del software per minimizzare i costi. Vi sono presentati e confrontati i modelli di Putnam, Brooks e Halstead e alcune tecniche di produzione e verifica del software. Il capitolo costituisce di per sè una panoramica rapida ma abbastanza esauriente, ideale per un primo accostamento a un argomento importante e oggetto di attive ricerche (il numero di marzo 1983 delle IEEE Transactions on Software Engineering, per

(*) Rileggendo più attentamente la prefazione scopro che l'autore afferma solo che il *programma* per l'hash perfetto è originale: in effetti è una grossolana semplificazione di quello presentato da Sprugnoli!

esempio, contiene due analisi critiche della teoria della software science di Halstead).

C'è poi soprattutto la seconda metà del testo, dedicata ai processi concorrenti: non ho trovato in nessun altro libro a mia conoscenza un'esposizione così ampia e dettagliata sui processi concorrenti. Direi che anche il livello della trattazione diventa più alto: mentre la prima parte non richiede particolari prerequisiti, la seconda non mi sembra adatta per un primo approccio all'argomento, proprio perchè l'autore si immerge presto nei dettagli e non sempre con spiegazioni sufficienti. Anche qui, poi, non tutto è limpido; per esempio la definizione di processo o le nozioni di processi concorrenti e paralleli; dapprima i processi paralleli sono presentati come un sottoinsieme dei processi concorrenti, subito dopo i due insiemi sono presentati come disgiunti. Le reti di Petri, strumento di elezione per trattare problemi di concorrenza, vengono usate poco più che per una funzione decorativa. Non essendo uno specialista di processi concorrenti, mi risulta abbastanza sorprendente vedere definiti termini propri (per me) delle reti di Petri, come sicurezza, lealtà e vivezza, senza riferirli alle reti stesse, e ho insomma l'impressione che queste ultime siano usate grandemente al di sotto delle loro possibilità.

Chiude il libro un capitolo sul software tollerante i guasti: come dice l'autore nel sommario, "la teoria della tolleranza ai guasti per il software è ancora alla sua infanzia". Il fatto che l'autore abbia voluto includere questo argomento torna a suo merito, e conclude significativamente un volume non privo di difetti, ma certamente stimolante e coraggioso per la scelta degli argomenti e il tentativo di introdurre nuovi approcci.

Mauro Torelli
Istituto di Cibernetica, Univesità di Milano.

Honeywell
Honeywell Information Systems Italia
Via Laboratori Olivetti - Pregnana Milanese

UNIVERSITA'
DEGLI STUDI DI MILANO
ISTITUTO DI CIBERNETICA
Via Viotti, 5 - Milano

Honeywell
Honeywell Information Systems Italia NOTESW2122 M011001 01

CIMINO MICHELE

HISI-DIREZIONE GENERALE MARKETING ITALIA

VIA PIRELLI,32
20124MILANO

Ufficio Documentazione Gestione Mailing List
Via Vida, 11 - Torre B - **20127 MILANO**
Trattasi di trasporto di cancelleria-stampati-moduli-documenti interni-mobili e macchine per ufficio ESONERATI DALL'OBBLIGO DELLE BOLLE DI ACCOMPAGNAMENTO D.P.R. 6/10/78 n. 627 art. 4 n. 8 e Circolare Ministeriale n. 72 del 23/12/78 paragrafo 10